University of Bath

PHD

Application of fault tolerant techniques to a real time control system.

Jackson, P. R.

Award date:
1983

Awarding institution:
University of Bath

Link to publication

Alternative formats
If you require this document in an alternative format, please contact:
openaccess@bath.ac.uk

General rights
Copyright and moral rights for the publications made accessible in the public portal are retained by the authors and/or other copyright owners and it is a condition of accessing publications that users recognise and abide by the legal requirements associated with these rights.

- Users may download and print one copy of any publication from the public portal for the purpose of private study or research.
- You may not further distribute the material or use it for any profit-making activity or commercial gain
- You may freely distribute the URL identifying the publication in the public portal ?

Take down policy
If you believe that this document breaches copyright please contact us providing details, and we will remove access to the work immediately and investigate your claim.

Download date: 11. May. 2021

APPLICATION OF FAULT TOLERANT TECHNIQUES TO A REAL TIME CONTROL SYSTEM

submitted by P.R. Jackson B.Sc.
for the degree of Ph.D.
of the University of Bath
1983

COPYRIGHT

Attention is drawn to the fact that copyright of this thesis rests with the author. This copy of the thesis has been supplied on condition that anyone who consults it is understood to recognise that its copyright rests with its author and that no quotation from this thesis and no information derived from it may be published without the prior written consent of the author.

This thesis may not be consulted, photocopied or lent to other libraries without the permission of the author for five years from the date of acceptance of the thesis.

.P. Jackson...

ProQuest Number: U641730

All rights reserved

INFORMATION TO ALL USERS
The quality of this reproduction is dependent upon the quality of the copy submitted.

In the unlikely event that the author did not send a complete manuscript
and there are missing pages, these will be noted. Also, if material had to be removed,
a note will indicate the deletion.

ProQuest U641730

Published by ProQuest LLC(2015). Copyright of the Dissertation is held by the Author.

All rights reserved.
This work is protected against unauthorized copying under Title 17, United States Code.
Microform Edition © ProQuest LLC.

ProQuest LLC
789 East Eisenhower Parkway
P.O. Box 1346
Ann Arbor, MI 48106-1346

UNIVERSITY OF BATH
LIBRARY
33
16 JUN 1983
PHD

CONTENTS

Figures.

Tables.

Synopsis.

The report describes a research investigation into fault tolerant strategies within a real time control system. Methods for increasing the reliability of a system other than through the use of fault tolerance have also been reviewed. The study which concentrated on a Recovery Block structure is separated into two parts, that is, a single and a distributed processing system. The single processor study involved modelling a subset of the control system; error recovery strategies are presented here as additions to the basic Recovery Block structure. Fault injection logic was specially designed and built in order that the recovery strategies could be tested under extreme operating conditions.

The distributed processing study is an extension of the single processor research. Three types of recovery are investigated to increase system availability; local recovery, global recovery and task swapping. The philosophy used in the distributed processing study was always to attempt recovery on a local basis, that is to prevent the propagation of faults to other microprocessors within the system. Global recovery is established as a method of maintaining continued safe operation when local recovery or communication between processors fails. The use of a standby processor system for dynamic task swapping is shown to give continued systems operation under conditions which would normally cause a catastrophic crash in non redundant systems.

The overall conclusion of the research is that fault recovery must be localised to prevent fault propagation from one process to the following process, with no distinction as to whether the communicating processes are in the same or different microprocessor subsystems, and that this can be successfully achieved in a real time environment by the use of a Recovery Block structure.

List of Symbols

ACP	Activity Channel Pool
A/D	Analogue/Digital
BC	Bus Controller
CPU	Central Processing Unit
D/A	Digital/Analogue
DMA	Direct Memory Access
EMP	Electromagnetic Pulse
FIFO	First In First Out
FMEA	Failure Modes Effects Analysis
I/O	Input/Output
LED	Light Emitting Diode
LSI	Large Scale Integration
MASCOT	Modular Approach to Software Construction Operation and Test
RAM	Random Access Memory
ROM	Read only Memory
RT	Remote Terminal
s-a-0	Stuck at logical '0'
s-a-1	Stuck at logical '1'
TMR	Triple Modular Redundancy
VLSI	Very Large Scale Integration
z	Discrete Operator
Z ( )	Z - Transform of ( )

List of Figures

Chapter 1. Introduction.

With the introduction of low cost sophisticated processing, the use of microprocessors has become an important part of the industrial scene, with LSI and VLSI devices often replacing analogue or large digital equipment. In addition to small size and high processing power, a microprocessor based system provides system flexibility with the capability of system reconfiguration. A growing realisation of the new problems that the change to microprocessors has brought about is now evident; the consequence of a system failure in applications such as satellite attitude control is severe, leading to a need for analysis and design techniques to be adopted in order to improve system reliability and availability. Such system failures can originate at either the design or manufacturing stages or in operational use. Design errors typically include systems analysis, hardware design, incomplete specification, mismatch of hardware and software, software design and coding. An analysis of program errors points to the fact that incomplete, inconsistent or ambiguous software requirement specifications are a significant problem.[1]

The reliability of a system may be improved by a combination of different techniques which fall into three main categories, fault avoidance, fault removal and fault tolerance. Chapter 2 reviews the considered techniques which are summarised below.

The avoidance of faults at the analysis and design stage can be carried out by the use of a formal specification language and associated design techniques. Fault tree analysis and failure modes effects analysis (FMEA) can be used to detect critical parts of the system; certain failure modes can then be eliminated at the design stage.

Fault removal techniques involve the construction and integrated testing of hardware and software prototypes. In addition the use of structured software enables a more thorough testing of the system to be carried out. The use of correctness proofs of software is beginning to emerge but is unlikely to replace prototype testing.

Fault tolerance is a further technique whereby redundant hardware and software is used for the protection and recovery from faults. The need for high reliability can be justified in systems where human life is at stake, where maintenance is not possible or in situations where a large financial loss results from a system crash.

1.1. Research Objectives.

A method ot increasing the availability of a given system is by the addition of redundant hardware and software to provide protection against and recovery from faults within and external to the system. It is important that the implementation of redundancy techniques is considered in terms of cost effectiveness, weight and power requirements; for example massive redundancy may not be a cost effective solution if only a marginal increase in reliability and availability is obtained.

The aim of the research study was therefore to investigate the possibility of increasing the availability of a given system by the inclusion of fault tolerant mechanisms for the protection and recovery from predefined faults. The aim can best be divided into constituent parts as follows:

(a) To establish good design practices based upon a practical rather than a mathematical approach.

(b) To establish a simple but obvious structure for system recovery.

(c) To establish design criteria for reliable inter-task communication within a single microprocessor system.

(d) To establish a design philosophy for message passing between microprocessors in a distributed system in order to inhibit the propagation of faults.

The research entailed an initial study of different strategies that could be adopted as a starting pcint. The next stage was to choose a system upon which the strategy could be applied. The aspects which govern system recovery under faulted conditions become more critical as the response time of the system decreases. With these factors in mind a decision was made to choose a real time system as opposed to a batch processing system since requirements for processing speed, criticality of system outputs and fault recovery time are much more demanding.

1.2. Research Model.

After careful consideration it was decided to base the study on a notional ground defence system, which consisted of a target tracking and missile guidance loop as described in Chapter 4, in order to establish the objectives previously mentioned.

The target tracking process consists of converting raw target data into a plot of target positions. The raw target data is produced from a radar whose aerial rotates at a constant rate, and consists of range and velocity data extracted from the returning radar signals.

The missile guidance loop consists of a proportional plus integral controller and the missile itself whose autopilot is represented by a second order function.

The tracking process determines the angular position of the target which is known as the target azimuth. This angle becomes the input for the guidance loop, whose objective is to constrain the missile to lie on a line joining the tracking system and the target.

The modelling of these functions in a microprocessor environment is described in Chapter 5.

1.3. Systems Implementation and Investigation.

A subset of the real time system was implemented on a single microprocessor to establish how well it was capable of detecting and recovering from faults within its system. The single microprocessor system carried out the function of the target tracking process with raw target data being provided from a PDP 11. The implementation of the target tracking process in a microprocessor system is described in Chapter 6. This system was then operated under fault conditions to provide a baseline for the results. Following this a fault tolerant structure was implemented; the results obtained from this are discussed in Chapter 7. The conclusions of the single microprocessor study are stated in Chapter 8.

Having gained valuable experience about the workings of the system under fault conditions, a distributed processing system was then investigated, this involved the choice of a communications link and the method for injecting real time faults onto the system; these topics are discussed in Chapter 9. The implementation of the complete real time system in a distributed processing environment is described in Chapter 10. The study also involved looking at a three processor system with protection and recovery methods for increased availability under fault conditions, the results of which are given in Chapter 11.

The use of a standby processor system is shown to give continued systems operation under conditions which would normally cause a catastrophic crash in non-redundant systems.

Chapter 12 looks at the question of when should such a redundant processor subsystem be used and presents results for the recovery of the system from the failure of a complete subsystem. The conclusions of the distributed processing study can be found in Chapter 13. This is followed by a review of guidelines for reliable systems design and the initial design of a single microprocessor system within Chapter 14.
A final chapter reviews the achievements made from the research study.

Chapter 2. Techniques for Reliable Systems Design.

The reliability of microprocessor based systems can be improved by a combination of several strategies: fault avoidance, fault removal and fault tolerance. The amount of work carried out in this area is considerable and this chapter summarises a number of techniques which are directed at enhancing the reliability of a system. In addition the problem of reliability prediction for microprocessor based systems is considered.

2.1. Failures, Errors and Faults.

To avoid ambiguity the terms failures, faults and errors are defined[2] and are used throughout this thesis.

Failure.

A failure of a system occurs when the system does not perform its service in the manner specified. This may be either because it is unable to perform the service or because the system outputs are not in accordance with the specifications. Thus a failure is an event.

Error.

An error is a part of an erroneous state which constitutes a difference from a valid state.

Faults.

A fault is the mechanical or algorithmic cause of an error. This encompasses areas of design inadequacies such as incorrect choice of component, system specification misinterpretation and incorrect interrelationship between system components (software and hardware).

2.2. Fault Avoidance.

The initial stage of a development process is the functional specification stage; this generally involves determining the requirements for both normal and abnormal operation of the system. Design faults that can arise during this phase include inconsistent requirements and misinterpretation or omission of requirements. Design inadequacies made during the requirements definition phase which are found at a later stage generally involve a redesign of software and/or system and repeat of the testing process. A reduction in the number of errors resulting could possibly be obtained by the introduction of formal system specification languages which serve as a communication aid between systems design, implementation and user. Research in this area is at an early stage; an example of a formal specification language can be found in Ref.3.

There exists a number of semigraphical methods for systems analysis. The most widely accepted of these methods is probably HIPO[4] (Hierarchy, Input, Process and Output) whereby functional specification is created by naming the basic functions which have to be performed and decomposing them into hierarchically ordered sub-functions. A further technique is the Structured Analysis and Design Technique[5]. This is basically a diagramming language which is used to describe the relationship between objects and activities. The amount of detail shown in a single diagram is controlled and thus leads to diagrams which can be quickly understood by management and users.

A technique gaining more acceptance is MASCOT[6] which provides a formalism for expressing the software structure of a real time system which can be independent of computer configuration and programming. It also provides a disciplined approach to design, implementation and testing of the system along with a strategy for documentation.

One of the most effective ways of avoiding design faults is to keep the complexity of systems design under control. Many software design methodologies based on this premise have been developed. They aim to structure software in a simple hierarchy of reasonably independent software modules. Work in this area includes reliable software through composite design[7] and the decomposition of systems into modules[8].

The use of small modules enables a complete understanding of their operation, in addition the consequence of modifications can be more easily seen than with one large program. Further, the use of structured programming leads to more reliable software and significantly improves the readability and maintainability of a module since structured code is read from top to bottom.

Consider now the hardware design; this is a task of selecting the most appropriate microprocessor and associated circuitry. A hardware/software trade-off has to be made, this is a matter of deciding which tasks are to be performed by software and which tasks by specialised hardware. Performing a task with specialised hardware incurs an extra cost in components and assembly for each product, whilst a software solution incurs a high development cost but has the advantage of non-recurring costs and ease of reconfiguration. A software solution to a problem will generally slow down the task execution unlike specialised hardware which can be designed to perform the task independently, for example a floating point arithmetic unit. Thus when difficulties arise in achieving the response time, then software should be replaced by hardware.

Systems design may be realised by a multi-processing solution since the processing power of a single microcomputer may be insufficient to meet the system requirements. In this case the software would be partitioned into independent tasks, each being located in the relevant subsystem. The communications protocol used between subsystems would then be determined by consideration of distance of transfer, data integrity and response requirements.

In applications where highly reliable systems are required, an analysis of failure modes is usually carried out following the design. An example of a technique for failure analysis is Fault Tree Analysis,[9] which starts by specifying a total system failure or safety critical failure. The analysis then proceeds downwards from this failure to identify part failure modes which could lead to such an overall failure. The final result is a highly detailed logic diagram depicting basic faults and events that can lead to the critical failure at the top of the diagram. Each basic fault is given a probability from an analytical or an empirical approach. The probability of the critical failure occurring is then calculated by appropriate means from probabilities of the basic part failures. This technique is often applied in safety analysis, particularly in situations where human life is at risk or where cost of failure is prohibitive, or where certain system failure modes must be eliminated at the design stage.

The choice of programming language is another consideration of reducting the number of design faults and several high level languages have been introduced to meet the demanding requirements of a real time system. The choice of a language is made by considering language facilities such as interrupt handling, I/O facilities, program structure inherent in language implementation, data structure appropriate to application, portability and efficiency of execution of obj ect code. Examples of this are languages such as Coral and RTL/2, which have been specifically designed for real time applications, although Coral suffers from lack of I/O facilities. Pascal has a good structure and is portable, whilst Concurrent Pascal has specifically been designed for multi tasking environments. Ada is still very new and may be too complicated to be reliable. In contrast PLM, PLZ and MPL have been specifically designed by Intel, Zilog and Motorola for their own chips and hence there is a lack of portability.

2.3. Fault Removal.

Despite efforts to avoid faults in the analysis and design stages, system failures will still occur due to residual design faults. Fault removal techniques can be applied during the design phase in order to remove as many of these faults as possible consistent with cost, development time scale and reliability requirements.

In the case of hardware many well proven techniques exist; these include design reviews, the building and testing of prototypes, inspection and testing of printed circuit boards and the use of component Burn-In to eliminate early failures.

The correctness of a systems design is important and must be checked before software coding is started. The use of structured walk throughs and design reviews are desirable where the correctness of each design step can be checked by the designer and project engineers.

A structured software system has the advantage that testing can be modular and more thorough thus removing a greater percentage of design faults. In top down testing, the top level is tested first, a lower segment is added and the combination tested. This is repeated down to the lowest level. Dummy segments temporarily replace the segment subordinate to the segment under test. These dummy segments can vary in complexity and may return constants or may be a primitive version of the segment being simulated. To enhance structured testing the length of a segment should be limited to a manageable level, say fifty statements to enhance readability and comprehension whilst minimising page turning. Usually each segment will correspond to one function and can be implemented as a procedure with a descriptive name corresponding to the function. Thus the limited size of segment in addition to single entry/exit, top to bottom flow of control makes programs easier to extend and maintain. Reliability is further enhanced because test plans for the segment are easier to specify and execute.

Techniques for formal proving of program correctness[10, 11] are unlikely to replace program testing, now or in the near future since there are many problems still to be overcome. It seems reasonable to doubt the ability of correctness proofs as it is difficult to write long programs without errors and program proving has so far been more difficult than the construction of programs. The solution may lie in the use of computer aids to check the proof or generate it. The problem that then arises is how do you check the proof checker. In addition, large program proofs probably have to be constructed of small modules which could lead to an interfacing problem between modules. The correctness proof must also include areas such as processor and system architecture, memory size and timing considerations.

2.4. Fault Tolerance.

Microprocessor based systems of the future are unlikely to be designed and built so as to be free from faults during their operational life.

Residual software design faults and random hardware faults are likely to occur; these must be detected, corrected and the system restored to a working state which leads to a need for built in redundancy for highly reliable systems operation. However, such redundancy must be applied carefully and in the correct structure, otherwise increased system hardware and software could lead to a reduction in reliability.

There are certain applications areas where use of fault tolerance is vital. First, there are systems where maintenance is not possible such as in space vehicles whilst reconfiguration around a malfunction may be possible. Secondly, fault tolerance is important in systems where human life is at stake, for example control of nuclear plants, ground defence systems and transport systems. Finally, there are applications in which computer downtime leads to financial losses such as automated process control and communication systems.

Having discussed the need for fault tolerance, consider now the types of faults that may occur during the operational life of the system.

2.4.1. Characterisation of Faults.

Faults occurring in a system may be attributed to a number of factors, e.g. temporary, intermittent or permanent failure of hardware components, hardware or software design faults or manufacturing faults. A fault causes an error if an incorrect state is entered; the fault does not always cause an error to occur immediately, for example a memory cell having a stuck-at 'logical 1' fault will not cause an error until a 'logical 0' is incorrectly read as 'logical 1'.

Temporary or transient faults are those of limited duration and can be caused by malfunctions of components or by the introduction of interference. If the duration of a transient fault is longer than a predetermined time then it will be interpreted as a permanent fault; for example a communications link may allow up to three re-transmissions of data before a permanent fault is reported.

Consider next the permanent failures of components; if the fault is not masked then it must be detected and recovery can then take place. This may consist of a software algorithm for hardware reconfiguration along with program and data rollback.

Local faults can be described as those that only affect a single logic variable whereas distributed faults are those which affect two or more variables. The advent of LSI and VLSI chips means that distributed faults are much more likely to occur than in the past, as a single gate is unlikely to fail without affecting other gates in a complex closely packed integrated circuit. Distributed faults can also be caused by failure of a single critical element, for example processor clock or power supply.

2.4.2. Redundancy Techniques.

The detection of a fault during operational use is the starting point of all fault tolerant mechanisms except those which use fault masking. In many systems it is important that these faults are detected quickly and are not allowed to propagate, otherwise system failure may occur.

In order to detect malfunctions the systems behaviour must be monitored in order to show deviations from the norm. This monitoring is generally performed by a combination of hardware and software techniques for detecting system malfunctions include the following:

(a) The pattern of states through which the system passes can be compared with expected or valid state transition patterns in order to reveal the presence of hardware or software faults.

(b) The performance of the system can be monitored to indicate fault free operation; this monitoring includes response time, system throughput and process calculation time.

(c) A malfunctioning system will often lead to the process trying to execute an invalid instruction or one that has an invalid address.

(d) The use of traps in processor software can be used to indicate, for example, division by zero or overflow conditions which may be caused by the propagation of a fault to the relevant instruction.

Hardware redundancy can be divided into two types, i.e. masking and standby redundancy as described below. Redundancy in the form of software is considered in section 2.4.4.

Fault masking is a technique widely used, whereby the fault is masked by the presence of additional hardware, the output remaining error free as long as the protection is adequate. One form of fault masking is the use of n - modular redundancy where majority voting takes place on the outputs of an odd number of identical units. The use of error correcting codes is another form of fault masking, the most common code being the Hamming code[12].

Standby redundancy can either be classed as cold or hot standby; the terms cold and hot relate to whether the redundant units are powered up. In cold standby redundancy, only one unit is powered up and operational, whilst the remaining units are not powered up. A schematic of cold standby redundancy is shown in Fig.2.1. A failure sensing and switchover device monitors the operation of the working unit and switches to one of the standby units when a failure of the working unit is detected.

In a hot standby redundancy scheme, all units are powered up, and are arranged typically as shown in Fig.2.2. This figure shows three units with the output of one of the units, chosen arbitrarily, providing the system output. If the comparator detects a disagreement, then the faulty unit must be identified and the system output taken from one of the other units. The time taken to switch from a faulty unit to a fault-free unit must be considered in the design phase.

2.4.3. Fault Recovery.

The detection of a fault provides the basis for the next step which is the correction and recovery of the system. Fault masking is a special case of system recovery which does not use separate fault detection.

In systems where high availability is necessary, the recovery from a fault must be automatic and not require human intervention.

Methods of recovery from a fault include:

(a) Re-try the operation that failed, if successful then continue.

This is particularly valid in the presence of temporary faults.

(b) Rollback of system to a position where system operation was known to be correct and repeat execution.

(c) Reconstruct or correct data structures from redundant data or status information.

(d) Re-initialise the system, with or without status information.

(e) Restore the system state to nominal or default values with the use of a status flag to indicate that output may contain inaccuracies.

(f) The use of standby spares either in a cold or a powered up condition.

System recovery can take one of three useful forms: full recovery, graceful degradation or safe shutdown. The techniques used in a particular system depend upon the extent of the damage, the possible cause of malfunction and the operating state of the system at the time of the fault.

2.4.4. Fault Tolerant Software.

The use of redundant elements is an established practice in fault tolerance of hardware. However, the use of redundant software for reliable operation requires special attention due to the nature of software. In contrast to hardware in which physical faults dominate, software defects are time invariant. Executing duplicate copies of a program in parallel does not improve the operation with respect to software defects, because software design faults will be inherent in both copies. The following paragraphs describe two methods of achieving fault tolerance in software: N -Version programming[13] and the Recovery Block[14].

2.4.4.1. N-Version Programming.

This approach is analogous to the well known hardware method of replication and voting on the outputs of the hardware modules. A number (N $\geqslant$2) of independently coded programs for a given process are run simultaneously on loosely coupled processors. The independent results are then compared, and in the case of a disagreement, a preferred result is generated by majority voting (for $N > 2$) or by a predetermined strategy. The success of this technique depends upon the level of independence that can be achieved in the N Versions of the program. Independence is best obtained by the use of different algorithms and programming languages in each version. Different data structures could also be used to increase the independence. The critical areas for this technique are the voting algorithm and the housekeeping prior to and after voting.

A constraint on N-Version programming is the requirement for N computers that are hardware independent, yet are able to communicate efficiently. The problem of synchronising arises here, a voter may have to wait for a result or indeed a result may never arrive due to a fault.

2.4.4.2. The Recovery Block.

This technique, in contrast to N-Version programming, can be applied to any configuration of processors, including a single processor. The structure in its simplest form is shown in Fig.2.3., where a process is described by a primary routine P. The output of the primary routine must pass an acceptance test T before passing control to the next process, if the acceptance test fails or if a set time has expired whilst executing the primary routine then a transfer to the alternate routine, Q, is initiated. If the acceptance test fails after execution of the routine Q or if a time out occurs during Q then an error return results. This technique does not preclude the use of several alternate routines if necessary for critical parts of the system.

It follows that a critical feature of the Recovery Block is the acceptance test. The alternate routines are worthless if failure of the primary routine is not detected by the acceptance test, thus the acceptance test must be thorough without being too time consuming.

A number of different types of acceptance tests are described in the following paragraphs:

(a) In many cases the definitions of the process imposes conditions which must be met at the completion of the process. These conditions can be used to construct the acceptance test. For example, an acceptance test for a sorting process may be to check the order, produced by the primary or alternate routines, is correct.

(b) Accounting checks can be used in acceptance tests for processes that are transaction oriented. The acceptance test could independently generate a checksum and compare it with the one produced by a primary or alternate routine.

(c) Another class of tests are called reasonableness tests. These tests are based on precomputed ranges of variables, expected sequences of program states or other occurrences that might be expected to occur in the system. Reasonableness tests are based on physical constraints whereas tests for requirements are based on mathematical or logical relationships. Tests used for acceptance can typically examine whether a variable is in range, whether the increment or decrement of a variable is in range or correlation between different variables is in range. For example, a process might calculate the acceleration of a missile. The acceptance test might simply test whether this acceleration is within predetermined limits, say ± 10g in order to maintain structural integrity.

(d) In an important process such as a firing sequence, the use of flags is a good way to ensure the correct procedure has been followed. In such a case, the acceptance test could check to see if all the appropriate flags have been set before firing is allowed to occur.

2.5. Reliability Modelling.

The reliability of microprocessor based systems has conveniently been divided into two areas, i.e. that of hardware and software, due to the two disciplines involved in the design. Hardware reliability modelling has been an established practise for many years whilst software reliability modelling has only made an appearance in the last ten years.

Consider first the modelling of software reliability.

2.5.1. Software Reliability Modelling.

Software has the unique property that it suffers no natural degradation, except in the special case of software stored on magnetic media. The purpose of an error prediction model is largely as a management aid to decide when enough testing has taken place and in assessing the confidence levels that can be placed in the software.

Many models that have been put forward use a bug counting approach. This approach has been used by Jelinski and Moranda[15] and by Schooman[16]. Jelinski and Moranda developed a software reliability model which assumes exponential distribution of faults and a software failure rate, i.e. the rate at which the software system fails to meet informal system requirements, which decreases in discrete steps as a function of time. Schooman's model is based on the same underlying assumptions with the difference that failure rate is also dependent upon the debugging effort. These models imply that reliability improvement can only take place at a system failure, since it is only here that a design error can be removed.

Musa[17] presents another model, using program execution time as the time variable rather than calender or debugging time as in the previously mentioned models. In addition he introduces a factor for non-corrections of the cause of the failure.

Schick and Wolverton[18] address the problem to a reliability model by determining an analytic stochastic model for predicting the number of remaining errors in the software, the mean time to next failure, the time to discover the remaining errors and the standard deviation associated with the error prediction.

Littlewood and Verrall[19] use a contrasting approach of no news being good news, where failure rate decreases between failures and periods of failure free working cause the reliability to improve.

Even if assumptions about failure rates being proportional to the number of errors remaining are accepted, then estimation of model parameters still poses great difficulty. One objective should be to measure the quality of the behaviour of the software, its operational reliability (integrity) rather than the number of design errors left in the program.

It is considered by the author that instead of establishing a figure for software reliability, in terms of number of remaining errors, that a range of software metrics be used for assessment of software integrity. This assessment must depend upon the compexity of the software modules, the criticality of each module to system performance, the tolerance of each module to errors caused by environmental factors and the maintainability and testability of the software.

Consider now the modelling of hardware reliability.

2.5.2. Hardware Reliability Modelling

The effects of environmental stressing are known as random failures. These failures occur in all types of electronic equipment and are generally treated as exhibiting a constant failure rate. This constant failure rate in non-redundant systems is supported by the use of life test and field data, after accounting for infant mortalities and the effects of maintenance.

In microprocessor based systems, malfunctions are dependant upon the component configuration, for example a failure may result from a transistor sinking excess current. Thus a reliability model must take account of prevalent failure modes.

The laws of probability govern the outcome of a mission of a redundant system and simple probability formulae clearly show the advantage of redundancy. Consider a triple modular redundant (TMR) system where three identical computers are used to give an output based on a majority vote. This system will only give an improvement in the mean time to error if maintenance is provided before the 'mean time before failure' of the individual modules. TMR systems are vulnerable to voting and single point timing failures which reduce the reliability of such systems.

Error detection and correction can be incorporated into integrated circuits to extend their 'mean time between failure' provided a comprehensive testing capability is also incorporated. An example of the design for testability of error correction circuitry for memory arrays is given in Ref.20. However, the effectiveness of any on chip redundancy will always be limited by the high correlation between malfunctions and the common thermal and structural failures.

In a complex system, the relationship between a random failure and its manifestation as an error is apt to be obscured by ill defined propagation paths. This is likely to cause problems for analytic models based on simple cause - effect relationships.

The modelling of some of the more complex redundant systems is often carried out by the use of Markov process models. These models can be made arbitrarily accurate by incorporating an arbitrary number of states. Caution must be applied in using these models on processes other than those with constant failure and recovery rates. A constant recovery rate is hard to imagine for a real time system as the time taken to recover depends upon configuration of the system at time of fault, the process being executed and the criticality of the fault.

Availability is measured as the percentage of time that a system is in an operational state. In some applications, the penalty for a single long unoperational period is much greater than that for many short periods, whereas the availability figure may be equal for the two instances. In this case, another parameter is required to describe the performance, i.e. time. This concept of penalising a slow recovery is discussed in Chapter 11.2.

Coverage of a system is the probability of the system recovering from a malfunction, it is a complex architectural attribute and is influenced by latency of fault, ambiguity in the perception ot the fault and by the architectural anticipation of such a fault. An estimation of coverage made before experimental verification is likely to be largely inaccurate. Retrospective coverage can be obtained but cannot accurately reflect any system other than that for which it was gathered.

Chapter 3. Analysis of a single Microprocessor System.

Having discussed techniques for reliable systems design in Chapter 2 an approach had to be chosen that could be used for single or distributed processing systems. A requirement of the research was that massive redundancy was to be avoided, if possible. The Recovery Block meets this requirement and in the view of the author was a good basis for further investigation, initially on a single microprocessor and then finally in a distributed processing environment.

In order to determine recovery mechanisms for a processor system under fault conditions, it became necessary to identify the effect of faults on system operation. An example of this identification is given here on a typical processor system consisting of CPU, RAM ROM, A/D and D/A convertors along with the necessary interconnecting and buffering logic, as shown schematically in Fig.3.1. The data bus transceivers, address and control buffers as shown in Fig.3.1. are permanently enabled and the direction of the data bus transceivers defaults to drive away from the CPU except when reading memory.

The approach of identifying failure modes and their effects is a useful method of fault avoidance. As hazards are identified, software and hardware defences can be developed using fault tolerant or self checking techniques to reduce the probability of their occurrence once the system has been implemented.

In the following section, typical causes and effect of faults are given for the described system; in addition possible solutions are given for the purpose of system recovery.

3.1. Cause and effect of Faults in a Typical Microprocessor System.

The following descriptions of causes and effects should be read with reference to Fig.3.1. The list is not exhaustive, but sufficient to identify typical fault effects in the view of the author.

	Cause	Effect	Possible Solutions
1.	No clock.	The system will stop.	The use of a fault tolerant clock[21].

	Cause	Effect	Possible Solution
2.	Address bit failure.	Incorrect addressing occurs resulting in CPU fetching data and/or instructions from wrong addresses.	A time out can be used to indicate that the program sequence was not completed in time. By monitoring of bus with other logic then it may be possible to re-arrange addressing of system, i.e. move program and data to another part of memory.
3.	Reset failure.	System fails to reset when required.	If reset fails then attempt to carry on processing.
4.	Read/$\overline{\text{write}}$ line.	If the line is stuck at logical '1', that is always a read cycle, then CPU is always reading from memory and I/O. When attempting to write then bus conflict will occur with CPU and memory buffers driving against each other. If the line is stuck at logical '0' then the system always sees a write cycle. When a CPU read cycle occurs then memory is loaded with garbage. The effect of an undriven bus will inevitably result in incorrect program execution.	A time out will indicate that a fault has occurred. Monitoring logic could give information on the nature of the fault.

	Cause	Effect	Possible Solutions
5.	Data bus transceivers.	If stuck at faults occur on the data bus, then bad data is read from or written to memory. If a fault in the direction logic occurs with direction always towards the CPU then bus conflict will occur; when writing to memory no data will be stored. If a direction fault occurs with direction always to memory, then when reading from memory the CPU will read a bus which is not driven.	There is a possible detection of an undriven bus as the CPU will probably read all 1's; alternatively the bus could be made to default to a particular instruction. A conflict on the data bus will cause time-out or a trap due to attempted execution of invalid instruction.
6.	Memory failure.	Incorrect instruction/data is read from memory.	The fault can be masked by automatic error detecting correcting codes, although CPU intervention or special logic may be needed to correct multiple faults.
7.	Address bus buffer.	As address bit failure.	See solution 2.
8.	Clock failure.	No memory accesses can be made.	The duplication of address and control buffers is possible but not cost effective.

Cause	Effect	Possible Solution
9. Valid Memory Address Signal.	If stuck at logical '1' fault occurs then memory is accessed at wrong point in time or spurious addressing occurs. If stuck at logical '0' fault occurs then memory is never accessed.	The effect is probably caught by a time out.
10. CPU.	The effects of such a fault are wide ranging and include stuck at faults on buses, invalid control signals and incorrect operations.	Repeated time-outs may possibly occur but CPU may not respond to them.
11. Address Decode Logic.	If no outputs from the address decode logic are enabled, then the CPU reads an undriven bus. If one output from the address decode logic is enabled, but it is the incorrect output then incorrect addressing occurs. If two outputs are enabled then memory is corrupted on a write cycle, and a bus conflict occurs on a read cycle. If the address decode logic is not enabled then no memory accesses will occur. If however the logic is always enabled then accidental addressing will probably occur.	There is a possibility of using self checking logic here.

	Cause	Effect	Possible Solution
12.	Buffered Read/$\overline{\text{write}}$.	As for effect 4.	See solutions 4 and 8.
13.	Memory Enable.	Bus conflict will occur if the enable occurs at the wrong time.	Possible solutions include self checking or monitoring by adaptive logic.
14.	Buffer for end of conversion of A/D convertor.	If the buffer is always enabled then bus conflict will occur. If the buffer is never enabled, then the CPU reads an undriven bus.	A bus conflict will probably cause a time-out in a program segment. If the buffer is never enabled then CPU will believe that conversion is not finished. The CPU could wait until conversion should have finished and then read the data. This data can then be compared with the last value to determine whether 'end of conversion' has not appeared due to a buffer or an A/D convertor fault. If an A/D convertor fault has occurred then set a flag and use another A/D convertor.

Cause	Effect	Possible Solution
15. End of conversion fault.	The conversion may appear to have finished early.	When polling to look for 'end of conversion' then check that it appears when expected and not before. The fault may be due to A/D convertor; use another A/D convertor if necessary.
16. Conversion command fault.	If accidental addressing occurs then an extra conversion command may be generated. However, the conversion command may not be given due to logic fault.	If accidental addressing occurs then an extra conversion will probably not matter. If no conversion command given then 'end of conversion' may not be cleared. The output of the A/D convertor can be compared with last value; switch to alternative A/D conversion if necessary.

	Cause	Effect	Possible Solution
17.	Data latch for D/A convertor.	If input or output lines of latch have stuck at type faults then incorrect conversion will occur. If the latch is not clocked then the last value clocked will be converted. If the latch is operated at the wrong point in time due to accidental addressing then an incorrect value will be converted.	The periodic connection of the D/A convertor output to the A/D convertor input could detect faults. If incorrect conversion occurred then CPU will detect the difference. If the latch is not working then the D/A convertor output will remain at last latched value and this will be detected by the CPU. If the latch is operated at wrong point in time then the D/A convertor output is neither correct (present) value nor last value and the CPU will detect this. If the data bus is not stuck then an alternative D/A convertor can be switched in.

3.2. Discussion of Failure Mode Effects.

The effects listed in the previous section for the faults considered are generally quite severe and continued system operation is unlikely if the faults are permanent. The most common of the effects appears to be incorrect addressing, leading to execution of the wrong instruction or use of the wrong data. The corruption of data within memory may occur even if memory is error correcting, since correction can only take place on faults within memory cells and not on incorrect data given to the error correcting memory.

The effect of faults on the control lines is similar to the effect of faults directly on the address and data lines. For example, a fault on the address strobe line may result in the wrong address being read or written to. This effect is similar to corruption of an address line, and may result in the microprocessor's program counter being corrupted.

If the faults are transient in nature, then the effects suggest that detection must include checking of data reasonableness, checking of address sequences and the use of the time domain for checking system operation. If permanent faults occur in a single microprocessor system, then continued system operation will not be possible in the majority of cases. Redundancy can be used to protect certain parts of the system, e.g. clock, memory and possibly the address decode logic.

Chapter 4. Real Time Systems Description.

This chapter describes a small real time system to be used as a basis for investigation into fault tolerant techniques. The system is complex enough to model a real system, but is simple enough such that complexity does not hinder the objectives of investigating the possibility of increased system availability. It was with this view in mind that the following operating characteristics were chosen.

4.1. Design Overview.

The system devised for the research investigation was a ground based target tracking and guidance process which selectively tracks a single target and determines whether the target is within missile coverage. The system is shown diagramatically in Fig.4.1. with an explanation of the component parts as follows.

The doppler radar consists of an aerial which rotates at a constant rate. The nature of this radar means that target information from a single rotation of the aerial is insufficient to determine whether the target is approaching or receding. The decision on whether a target is approaching or receding is made using information from successive scans of the aerial. In addition, the target tracking process determines whether the target is within missile coverage, i.e. has a missile a high probability of reaching and hitting the said target.

An operator can interact with the target tracking process and enter the system into one of two modes, i.e. search or track modes. The former mode of operation is used whilst waiting for a target detection.

The target tracking process constantly updates the azimuth on which a target lies; thus azimuth is referred to as Theta Beam in Fig.4.1. In order that only one target is tracked, the system uses an inhibition mechanism whereby target detection is only considered within a window around the last detected target position.

The target angle (Theta Beam) is used as the input to the missile guidance loop; this loop is stabilised by a digital controller using proportional plus integral control with a phase advance network. The digital controller generates an output proportional to lateral acceleration (latax) demand which is transmitted to the missile, which in turn produces a lateral acceleration as a result of this guidance demand. The guidance loop is closed by a simple relationship between the acceleration and the missile angle. Consider first the target tracking process.

4.2. Target Tracking.

This section describes the requirements of a target tracking process which processes target aircraft data and determines whether the target is within missile coverage. If a target is present on the same azimuth as the radar, which scans through 360° in one second, then it appears in a range/velocity channel. Target detection in a given channel defines the range and velocity limits within which the target lies. The detection of a target in a channel sets a pair of binaries; other binaries cannot become set until the original pair have been reset. An alarm is then set depending upon which pair of binaries has become set.

The azimuth and range at which a target is detected are used for inhibition purposes on subsequent scans and provide control for setting binaries. Due to the nature of the radar supplying target data, the system must decide whether the target is approaching or receding and use this information to determine whether the target is within missile coverage.

4.2.1. Target Data Input.

Data input to the system consists of six range and four velocity gates, giving a total of 24 channels. The range and velocity gates are combined by means of a matrix, shown in Fig.4.2. Some of the gates are arranged not to give an alarm, these correspond to slowly approaching or fast receding targets at maximum range. The combination of range and velocity gates which do not give an alarm are known as taboo channels and are shown diagramatically in Fig.4.3.

4.2.2. Azimuth Inhibit.

Following a target detection, the target position is stored in terms of azimuth,and range and velocity gates set. On subsequent scans a target will only be detected if its azimuth position lies within ± 24 degrees of the stored target azimuth, which moves with each target detection. The azimuth inhibit persists for four scans after the last detected target. The principle of the azimuth inhibit is shown in Fig.4.4.

4.2.3. Range Inhibit.

When a target is detected the target range is stored; on the two scans following this detection the system will only detect targets at the same range or within one range gate on either side of the stored target range.

4.2.4. Approach/Recede Identification.

The identification of the target as approaching or receding is carried out by examining range and velocity data from successive scans.

In search mode only one missed scan is allowable before the approach/recede decision is restarted, whereas up to four missing scans are allowable in track mode. The decision is based on four criteria as follows:

1. New target detection

   A new target is deemed to be approaching until a complete evaluation is completed.

2. Crossing target detection.

   A crossing target is defined as a target whose component of velocity towards the radar is close to zero.

3. A changing target range pattern .

   A target which has a rapidly changing range pattern is quickly identified as approaching or receding.

4. Doppler derived criteria .

   If a target remains within a given range gate for a number of scans then velocity gate information is used for the approach/recede assessment.

The algorithms for each of these criteria are not described in this thesis.

4.2.5. Missile Coverage.

Following the approach/recede algorithm the system identifies whether a target being tracked is within missile coverage. An 'in cover' indication represents a high probability that a target can be successfully reached by a missile. The determination of the coverage is described below.

4.2.5.1. Search Mode.

In search mode, 'out of cover' is indicated if the target is deemed to be receding and the angular rate appropriate to the alarmed range and velocity gate is zero. Table 4.1. shows the angular rate information for range and velocity gate combinations.

4.2.5.2. Track Mode.

In track mode, Table 4.1. is used to determine whether the target is in or out of missile coverage for the appropriate range and velocity gate combination. If the angular rate, calculated as below, is less than the value in lookup table, then 'in cover' is set, otherwise 'out of cover' is set.

Angular Rate = 100 -(10 x Number of alarms on target)..... (4.1.)

Having described the target tracking process, now consider the missile guidance loop.

4.3. Missile Guidance Loop.

The guidance loop used is a line of sight guidance loop where the missile is constrained to lie as nearly as possible on the line joining the defence system and the target.

The position of target is identified by a scanning radar aerial which rotates once per second. The target tracking process described in the previous section provides the position of a single target. The azimuth position of the target being tracked is then used as the input to the missile guidance loop which is taken from Ref.22 as shown diagramatically in Fig.4.5. This consists of a controller, missile autopilot and a double integrator for kinematic loop closure.

The controller consists of proportional plus integral control with an integrating time constant of two secs. In addition a double phase advance network, giving a maximum phase advance of 62.6° is used for loop stabilisation.

The missile autopilot is represented by dynamics defined by a natural frequency of 12 rads^{-1} and a damping factor of 0.6. The missile produces a lateral acceleration as a result of a guidance demand. Kinematic loop closure of the guidance loop from lateral acceleration to position results in 180° phase lag represented as a double integrator.

The Bode plot for this loop is shown in Fig.4.6. giving a phase margin of 35° and a gain margin of 10.5dBs. The step response of the analogue system is shown in Fig.4.7. giving an overshoot of approximately 55%.

Chapter 5. Modelling of Real Time System.

The system described in Chapter 4 consists of two distinct parts: the target identification process and the guidance loop. In order to model this system, it became necessary to simulate a target being tracked by a radar. This chapter describes how the above processes were modelled in order to represent a realistic real time control system.

5.1. Target Simulation.

Target simulation is performed by a program which was specifically written for this study to run on the PDP 11. The program is designed to handle multiple targets, but for the purpose of this study only a single target was considered. The target is characterised by a start co-ordinate (x, y, z), a heading co-ordinate (s, t, u) and a velocity; a straight line course is assumed between the two co-ordinates. The range of the target from the tracking system, situated at (0, 0, 0) is given by equation (5.1.) assuming the target is at co-ordinate (a, b, c)

$$\text{Slant Range} = (a^2 + b^2 + c^2)^{\frac{1}{2}} \qquad .....(5.1.)$$

The target is then tracked by a radar whose characteristics are given by:

Measurable Slant Range:	1 Km to 7 Km
Measurable Velocity:	50 m/s to 450 m/s.

A complete revolution was initially divided into 30 equal segments. If a target is seen in the aerial's beamwidth at a particular point in time then the appropriate range and velocity gates are set.

Thus for each 1/30th second the program gives an output of six range and four velocity gates, either set or unset as determined by the target position. Ten complete scans are simulated, representing ten seconds of target motion. This duration was chosen as this period of results of the target tracking process conveniently fills the temporary storage available.

The target chosen for the first part of the study has the following characteristics:

Start Position:	800	1500	200
Heading:	-100	1400	190
Velocity:	400		

The units for the start position and heading are metres whilst the velocity is in metres/second. This target was chosen as it represents a crossing target, i.e. the target is lost by the radar for approximately two seconds due to the fact that after about five seconds from the start of the run the target's component of velocity towards the radar aerial is close to zero.

5.2. Target Tracking Process.

The target tracking process consists of seven tasks interconnected as shown in Fig.5.1., which is a top level diagram of an SADT (Structured Analysis and Design Technique) activity model[5].
The tasks are described briefly below followed by typical results of the process.

5.2.1. Read Routine.

The read routine reads range and velocity data every 1/30th second. This data is precomputed by a simulation program and is stored in an area of microprocessor memory. If a target is detected, i.e. if any gates are set then the appropriate range and velocity channel variables are set to the appropriate values and the 'target detected' flag is set. The radar azimuth position is updated when the read routine is entered and can take values from 0 to 29. A flow chart of this routine is shown in Fig.5.2.

5.2.2. Process Azimuth Inhibit.

On the four scans following a target detection, the system considers targets only within a given angle (± 24°) of the last azimuth on which a target was detected. A flag, 'target azimuth valid' is used to signify if a target has been detected within the last four scans.

A flow chart of this routine is shown in Fig.5.3.

5.2.3. Process Range Inhibit.

On the two scans following a target detection, the system considers targets only within ± 1 range gate of the gate set when the target was detected. If 'azimuth inhibit' is set at any time then 'range inhibit' is also set. A flag 'target range valid' is used to indicate if a target has been detected within the last two scans.

If more than two missing scans occur then 'target range valid' is set invalid awaiting a new target, or reappearance of an old target. A flow chart of this routine is shown in Fig.5.4.

5.2.4. Set Binaries.

The set binaries routine decides which pair of binaries (if any) becomes set; only one pair of binaries can be set at any one time. Another pair of binaries cannot become set until a target appears in a range/velocity channel and the 'range inhibit' is not present. The setting of new pair of binaries resets the old pair. A flow chart of this routine is shown in Fig.5.5.

5.2.5. Process Binaries.

The routine determines if the pair of binaries set are allowed to generate an alarm. This is performed by the use of a look up table of taboo channels.

Two types of alarm can be generated; internal and external. The internal alarm is used for control of the approach/recede and coverage assessments whilst the external alarm is an indication to the operator. The external alarm is given to the operator only in search mode. A flow chart of this routine is shown in Fig.5.6.

5.2.6. Approach/Recede Assessment.

The approach/recede algorithm in track mode is different from that performed in search mode, as previously described in Chapter 4.2.4. Before the algorithm is started several other variables are calculated, these include the number of scans at the same range, variations in range between successive scans and identification of crossing targets. A flow chart of this routine is shown in Fig.5.7.

5.2.7. Coverage Assessment.

The coverage assessment is based upon a look up table which determines whether the target is in or out of missile coverage. The entry within the table is identified by the particular range/velocity binary pair set and whether the target is deemed to be approaching or receding.
If no binaries are set then the previous coverage indication remains for four aerial scans or until a new pair of binaries become set when coverage is reassessed. A flow chart of this routine is shown in Fig.5.8.

5.2.8. Baseline Performance.

Using the target characteristics given in Chapter 4.1., the target tracking process was run for ten seconds to provide a baseline performance. Fig.5.9. represents some of the outputs of the target tracking process.

An explanation of these graphs follows:

Fig.5.9(a) Azimuth Position. This represents the internal radar azimuth position; the ramp up to 30 represents the rotation of the aerial through 30 sectors of 12 degrees each.

Target Detected. This is a flag used to inform the system that a target has been detected, i.e. a combination of range and velocity gates have been set. The absence of the flag at five seconds is due to the crossing target.

Target Azimuth. The target azimuth is a record of the current azimuth on which the target being tracked lies. This variable is used for azimuth inhibition if 'target azimuth valid' is set. The target being tracked changes from appearing early in the aerial scan to late in the aerial scan as it moves from right to left across the sky.

Fig.5.9(b) Range Inhibition. Information on the target is updated only when range inhibition is not set. No information on the targets range and velocity is updated during the period of crossing.

Binaries Flag. This flag is used to identify whether the last stored pair of range/velocity binaries are valid.

Internal Alarm. This informs the system that a target has been detected within the last second. The alarm is set to zero at about five seconds due to the crossing target, although the system still remembers the target as up to four missing scans are allowed. The alarm is set again when the target reappears after approximately two seconds.

In Cover. This graph shows that the target being tracked is deemed to be within missile coverage.

5.3. Missile Guidance Loop.

In order to implement the guidance loop on a microprocessor system, it became necessary to digitise the transfer function. From Fig.4.6.a., it can be seen that the analogue crossover frequency is 3.4 rad.S^{-1}. A sampling frequency had to be chosen that was a compromise between a low sampling frequency resulting in aliasing and a high sampling frequency where inaccuracies occur due to finite word length. The sampling frequency chosen was 30 Hz which conveniently ties in with the 30 sectors in 360° for the target tracking process. The guidance loop, Fig.5.10. was implemented on two microprocessors, one processor performing the digital controller process and the other simulating the missile autopilot. Thus in digitising the complete guidance loop it is necessary to include two zero-order hold circuits as shown in Fig.5.10. Combining the missile autopilot with the kinematic loop closure, the guidance loop consists of two separate parts. Z Transforms were used todigitise the two separate parts.

From Fig.5.10.

$$G_1(z) = (1 - z^{-1}).Z\left(\frac{1}{S}.\ \frac{10(S+1)(S+1)(S+0.5)}{S(S+3.16)(S+3.16)}\right) \quad \ldots\ldots (5.2.)$$

and

$$G_2(z) = (1 - z^{-1}).Z\left(\frac{1}{S}.\ \frac{144}{S^2(S^2+14S+144)}\right) \quad \ldots\ldots (5.3.)$$

The transfer functions of the controller and the missile in terms of z^{-1} is derived by taking partial fractions, and then Z Transforms of the component parts, along with setting T = 1/30 second. A full derivation of $G_1(z)$ and $G_2(z)$ can be found in Appendix A.

This results in the following equations:

$$G_1(z) = \frac{10(1 - 2.918785963z^{-1} + 2.839590856z^{-2} - 0.9207881866z^{-3}}{(1 - 2.800048928z^{-1} + 2.610092963z^{-2} - 0.810044035z^{-3})} \quad \text{.....(5.4.)}$$

and

$$G_2(z) = \frac{-0.000903747z^{-1} + 0.002798632z^{-2} - 0.002670325z^{-3} + 0.0009156z^{-4}}{1 - 3.500869446z^{-1} + 4.628827977z^{-2} - 2.755048263z^{-3} + 0.627089085z^{-4}} \quad \text{.....(5.5.)}$$

Having derived Z Transforms for each of the two parts of the system, it is necessary to transform these equations into difference equations so that they can be executed on a PDP 11 or a microprocessor.

5.3.1. Floating Point Arithmetic.

The guidance loop was initially modelled on a PDP 11 using floating point arithmetic with seven significant decimal figures. Floating point arithmetic was used to determine the best realisation of the Z Transform equations before proceeding to execute the difference equations on a microprocessor with integer arithmetic. Three realisations were used and these are described in the following paragraphs.

5.3.1.1. Direct Realisation.

The first realisation used the Direct method for transferring the transfer functions in z^{-1} into difference equations.[23] Given that

$$\frac{U(z)}{E(z)} = \frac{a_0 + a_1z^{-1} + a_2z^{-2} + a_3z^{-3}}{1 + b_0z^{-1} + b_1z^{-2} + b_2z^{-3}} \quad \text{..... (5.6.)}$$

then by the Direct method

$$U_n = a_0E_n + a_1E_{n-1} + a_2E_{n-2} + a_3E_{n-3}$$
$$- b_0U_{n-1} - b_1U_{n-2} - b_2U_{n-3} \qquad .....(5.7.)$$

Inserting the coefficients of equation 5.4. into equation 5.7. gives the following difference equation for the digital controller

$$U_n = 10E_n - 29.18785963E_{n-1} = 28.39590856E_{n-2} - 9.207881866E_{n-3}$$
$$+ 2.800048928U_{n-1} - 2.610092963U_{n-2} + 0.810044035U_{n-3} \qquad ..... (5.8.)$$

Likewise inserting the coefficients of equation 5.5. into equation 5.7. gives the following difference equation for the missile

$$Y_n = - 0.000903767U_{n-1} + 0.002798632U_{n-2} - 0.002670325U_{n-3}$$
$$+ 0.0009156U_{n-4} + 3.500869446Y_{n-2} - 4.628827977Y_{n-2}$$
$$+ 2.755048263Y_{n-3} - 0.627089085Y_{n-4} \qquad ..... (5.9.)$$

The guidance loop step response for this realisation is shown in Fig.5.11. and gives an overshoot of 61% with a settling time of approximately six seconds to within 1% of the final value.

To ensure that the simulation was not conditionally stable, the binary representation of the coefficients was carried out. This representation is necessary for the realisation of the loop in integer arithmetic.

The resolution was set such that the smallest number which could be represented was 2^{-11}. The simulation was again run with a unit step input and the output is shown in Fig.5.12. This shows that the direct realisation of the guidance loop is unstable with a binary representation of the coefficients.

5.3.1.2. Cascade Realisation.

The second approach was to use the Cascade method of realisation. In this method the transfer function is expressed as a product of simple block elements.

$$\frac{U(z)}{E(z)} = a_0 D_1(z) \cdot D_2(z) \quad \ldots\ldots \quad Dm(z) \qquad \ldots\ldots(5.10.)$$

where m is less than n, the order of the system; a_0 is a constant. The block elements consist of either first or second order elements. Using this method, the guidance loop was divided into block elements as shown in Fig.5.13. The response to a unit step input is shown in Fig.5.14. and is similar to that of the direct realisation shown in Fig.5.11. The binary representation of the coefficients using the cascade realisation was carried out using the same resolution as above and this gave a step response as shown in Fig.5.15. This shows slightly less overshoot than for the realisation with exact coefficients (Fig.5.14.).

5.3.1.3. Parallel Realisation.

Finally the Parallel method of realisation was used to simulate the guidance loop. In this method, the transfer function is expressed as the sum of parallel units which are either first or second order, i.e.

$$U(z) = a_0 + D_1(z) + D_2(z) + D_m(z) \qquad \ldots\ldots(5.11.)$$

where m is less than n, the order of the system; a_0 is a constant. Using this method, the guidance loop was divided into elements as shown in Fig.5.16. Applying a unit step input, the output settles as shown in Fig.5.17. The response shown in Fig.5.18. represents the same realisation, except that the coefficients have been binary rounded as above.

Under no fault conditions, the parallel and cascade structures give similar results, however under conditions of a fault in a basic element, the cascade structure suffers from the fact that a fault is multiplied by each successive unit. The direct realisation was unstable with binary rounded coefficients and was left out of any further analysis.

5.3.2. Integer Arithmetic.

Having obtained stable results from both cascade and parallel realisations of the guidance loop, the next step was to perform the difference equations in integer arithmetic on a microprocessor, initially in 16 bit arithmetic. Using a unit step input the parallel realisation gave the output shown in Fig.5.19. and shows an overshoot less than 50%. However when the error signal becomes small the output shows quantisation errors. The 16 bit cascade realisation, whose output for a unit step input is shown in Fig.5.20., suffers from quantisation much more than the parallel realisation. The output is completely unsatisfactory and shows that this realisation has no practical use, and was therefore discarded.

In order to improve upon these results, the software for the two realisations was converted to perform 32 bit integer arithmetic. To increase the sampling frequency at this stage would only have increased the quantisation due to a finite word length. Using the same input as before, both realisations (Fig.5.21. and 5.22.) show improved responses which agree with that of the continuous system shown in Fig.4.7.

From the above results obtained, the parallel realisation of the guidance loop using 32 bit arithmetic was chosen as the 16 bit cascade realisation gave poor results and 'cascades' any error caused by hardware or software.

Chapter 6. Implementation.

The system used in this study to assess the effectiveness of redundant software and hardware for fault detection and recovery is based on a single Z8000 microprocessor. This processor was used throughout the research study and a description can be found in Appendix B. The Z8000 is connected to a Micromaster (Appendix C refers), via a serial link, which is in turn connected to a PDP 11/34. This chapter describes the hardware and software which was designed and completed for this study.

The software for the target tracking process is assembled on the PDP 11 and then transferred to the Micromaster be fore being loaded into the memory of the Z8000, as shown in the systems diagram in Fig.6.1. Assembler code was used in order to effectively monitor the effects of faults upon system execution.

The Micromaster acts as a terminal for the PDP 11 and controls flow of programs and data to and from the Z8000 processor system, which is situated on an Am96/4016 Evaluation Card[24]. In order to inject faults onto the processor system, the processor buses are brought out from this card into an expansion box shown in photograph Fig.6.2.
The expansion box contains the manual switch arrangement for the injection of faults onto the processor bus, in addition to system memory and input/output. The memory and I/O maps were designated as shown in Fig.6.3. and 6.4. respectively.

6.1. System Memory.

The Evaluation card contains 8K bytes of dynamic RAM which is used to store the target data.

Memory organisation in the expansion box is such that any one RAM chip is assigned to only one bit of a word in memory so that a memory failure (either single cell or complete RAM) will not cause more than one bit to be in error.

The error correcting memory, shown schematically in Fig.6.5. is situated on two double Eurocards. The first card consists of the data memory, whilst the second consists of parity memory, error code generation and the error correction circuitry. The two circuit diagrams are shown in Fig.6.6. and 6.7. respectively, whilst the layout diagrams and parts list are shown in Figs 6.8. and 6.9. and Tables 6.1. and 6.2.

The operation of the error correcting memory is briefly described for both the read and write conditions as follows. Consider the operation of writing to memory. The data word is written directly into the data memory whilst the parity bits are generated from the data bits by a set of parity equations and are written into parity memory.

On a read operation, the data word is read from memory along with the associated parity bits. Parity is then regenerated from the data word. If an error has occurred in a memory cell that is being read, one or several of the parity bits will be in error. The parity bits are then decoded to determine which data bit is in error. The erroneous data bit is then corrected by the exclusive OR operation and is buffered onto the data bus by an inverting buffer. Note that the exclusive OR operation inverts all bits except the bit in error (if any). The correct polarity is restored by the use of the inverting buffers, as shown in Fig.6.5.

6.2. System Input/Output.

System inputs can be divided into two types, firstly target data which is produced on the PDP 11 and down loaded via the Micromaster. Secondly inputs are provided in the form of switches on the front panel of the expansion box; these inputs represent the mode of operation of the tracking system and a system cancel facility. System outputs are in the form of LED's and consist of an operator alarm, an error signal and indications to inform the operator that a target being tracked is within missile coverage. The circuit diagram layout diagram and parts list of the input/output card are shown in Figs.6.10. and 6.11. and Table 6.3.

The expansion box also houses a buffer card which buffers all signals from the Z8000. The circuit diagram, layout diagram and parts list of this card are shown in Figs. 6.12. and 6.13. and Table 6.4.

6.3. System Software.

The system incorporates a suite of programs which are required for the various tasks involved; these being shown diagramatically in Fig.6.14. With the exception of the PDP 11 graphics and plotting routines this software was developed by the author for this study. The software is explained by means of following a typical run to generate ten seconds of system results. A flow diagram of the software is shown in Fig.6.15.

Initially fault data is produced by generating exponentially distributed fault interval times and uniformly distributed faults across the address and data bus.

Following this, the target data is generated by the target simulation program described in Chapter 5.1. A Z8000 cross assembler was written to generate assembly code listings and object code files for the target tracking process. The cross assembler runs on a PDP 11 and is described in detail in Appendix D. The object code file produced, approximately 3K bytes in size for the target tracking process, is transferred first to the Micromaster and then loaded into memory in the expansion box. The target data takes the same path to the Z8000 system and resides in the memory on the Evaluation Card.

The target tracking process, described in Chapter 5.2. is then run by commands to the Z8000 monitor via the Micromaster Keyboard. The injection of faults is carried out during the operation of this software and is described in the next section. Results are periodically sent from the Z8000 to the Micromaster and are stored there until the end ot the ten second run, when they are transferred to the PDP 11 and written into a disk file. The disk file contains blocks of data which can then be sorted in a form ready for the plotting routines. Fig.5.9. shows a typical set of graphs produced in this manner.

6.4. Fault Injection.

Two alternatives existed for the injection of faults; these are as follows:-

(a) Injection of faults within each of the memory and I/O devices connected to the buses

or

(b) Injection of faults directly in the buses which are common to all memory and I/O devices.

The second of these alternatives was chosen as it simplified the circuitry required together with providing greater flexibility.

Implementation of the fault injection logic was achieved by intercepting the buses by a logic and switching circuit. This arrangement allowed up to two bits of each of the address and data buses to be injected with faults at any one time. The faults can be stuck at logical '0' (s-a-0), stuck at logical '1' (s-a-1) or open circuit.
Two timers were used so that faults injected onto the data bus could be of different length to those on the address bus, and to ensure that faults are injected onto the respective buses at the appropriate time in the cycle.
The design of logic to inject faults with the previously mentioned properties is described in the following section.

6.4.1. Design of Fault Injection Logic.

Consider the design of fault injection logic for the single direction address bus. The requirement for the logic was that the output be:

1. As input
2. s-a-0
3. s-a-1
4. Open circuit

The selection and injection of these conditions is shown diagramatically in Fig.6.16., where for simplicity of presentation a single pole switch selects either a fault or no fault condition. Consider initially the first three requirements listed above, for the purposes of design let A be the input

of the block (see Fig.6.16.) and Z be the output of the block.
The fault control consists of two inputs, one to decide if a fault is to be applied (to be known as X) and the other to decide whether the fault is stuck at logical '0' or '1' (referred to as Y).

The control input is defined as: X at logical '0' - No fault
X at logical '1' - Inject fault

The type of fault injected is determined by the condition of the Y input which is defined as: Y at logical '0' - s-a-0 fault
Y at logical '1' - s-a-1 fault

Constructing a truth table it follows that the output of the block is given by: $Z = A\overline{X} + AY + XY$

$$\ldots\ldots(6.1.)$$

Since three input OR gates are not available, the equation 6.1. was rewritten using De Morgans law to give:

$$Z = \overline{\overline{A\overline{X}}.\ \overline{AY}.\ \overline{XY}} \qquad \ldots\ldots(6.2.)$$

which can be implemented as shown in Fig.6.17.
The output Z can then be optionally open circuit by adding a tri-state bus driver. Circuitry providing the control input (X), the condition input (Y) and the tri-state buffer driver (disable) is shown in Fig.6.18.

Two of the above circuits were built so that up to two faults can be injected onto the systems address bus. The two lines which are injected with faults are switch selectable; the switching arrangement is shown in Fig.6.19. which also shows the switching for the data lines. The switches are shown in a position representing a typical fault injection path.

Since the data bus is bi-directional it required more logic to implement fault injection compared with the address bus. This was accomplished by using two of the circuits shown in Fig.6.17., back to back with a direction select (READ/WRITE) as shown in Fig.6.20.

A wait state is used to extend the memory access, as the circuitry described above incurred a delay of approximately 40 nanoseconds.
The length of the applied faults is adjustable, via potentiometers on the

expansion box front panel, between approximately 100 nanoseconds and 1000 nanoseconds; although this does not preclude the possibility of leaving the fault on for any number of instructions. The length of the open circuit fault is not adjustable and was fixed on a per instruction basis; this was thought to be a flexible enough arrangement.

The fault injection logic is mounted behind the front panel on the expansion box close to the fault selection switches, as shown in photograph Fig.6.21. The circuit diagram, layout diagram and parts list can be found in Figs 6.22. and 6.23 and Table 6.5.

6.4.2. Method of Fault Injection.

The procedure for the injection of faults onto the processor system is as follows: the system is run for X instructions where X is an exponentially distributed variable. This type of distribution was used as it is typical in reliability studies. The system then halts and the fault is set up on the front panel; the fault being introduced onto the processor bus when a single step command is given. Although the system is not run at full speed, it was time scaled to ensure that recovery from a fault takes place within a given time. When the fault has been injected, the system is run for another Y instructions, where Y is another exponentially distributed variable with the same mean as above. It was decided that 90% of the faults should be of s-a-0 or s-a-1 type, with the remaining 10% being open circuit faults. A uniform distribution was used to determine which bit of the address bus or the data bus was to be faulted.

6.5. System Integration and Test.

In conclusion to the chapter, system integration and testing was carried out to establish that the design requirements had been satisfied. These tests were extensive and consisted of procedures specially developed, but which have not been included in this thesis.

Chapter 7. Design Strategies: Single Processor System.

This chapter presents strategies for detection and recovery from transient hardware faults, their implementation on a single microprocessor system and their performance under extreme operating conditions.
The approach taken here was initially to inject faults on the target tracking system with no recovery mechanisms to provide a baseline for the results. Having obtained a baseline, the next step was to use the basic Block Recovery structure and then build upon that structure to provide a recovery mechanism for a greater proportion of faults.

The software described in this chapter was stored in RAM as this gives less protection against faults than if the software were held in ROM; thus results obtained are worst case, since program memory is not write protected.

It was decided that a system run should last ten seconds, as previously explained and that during this time a large number of faults would be injected in order to keep down the number of runs. It was decided that the interval between faults be exponentially distributed such that the mean number of faults that were injected was thirty.

7.1. System with No Recovery.

In order to obtain a baseline set of results, the system was initially operated without any recovery or protection software or hardware.
The criteria for improved systems availability taken here was the percentage of runs that successfully complete ten seconds of operation, to produce valid outputs at the end of that time.

A total of twenty runs were carried out, of which only one was successfully completed. A further four runs completed the ten seconds, but did not give correct outputs during that time. Without any protection, system variables were often corrupted due to faults injected and these faults were allowed to propagate unchecked. Once a variable has been corrupted then, due to the lack of acceptance testing,it is passed onto the next process which will also give incorrect outputs.

The most surprising effect of faults was that only a minority caused the program counter to be badly corrupted immediately following the fault contrary to intuition. Given that the program counter was at address X before fault, then the effect of the majority of the faults was to leave the program counter within the range X ± 256 bytes. This is due to the small percentage of the total instruction set that allow a large deviation from the present program counter. Instructions that allow this large deviation include jump to absolute address, call subroutine with absolute address and reload program counter from memory. The consequence of the program counter generally staying local immediately following a fault is that it is not necessary to separate primary and alternate routine software into separate blocks of memory with the provision of enabling and disabling memory, but that it is sufficient to separate the two routines by a trap area of 256 bytes.

Typically a fault can lead to execution of wrong instructions, due to either an address or data fault. After one of these instructions the program counter is often set to a non-instruction word boundary. This then leads to corruption of register contents or a misinterpretation of instruction. Although the fault may only have occurred for the duration of a single instruction, an instruction word boundary may not be reached for several instructions.

7.2. Basic Recovery Block.

Having established the baseline, the next step was to implement the Recovery Block on the target tracking software. For each process, an acceptance test and an alternate routine was devised for checking and standby purposes. A brief description of this software can be found in Appendix E. The acceptance tests used here were fairly simple consisting typically of checking that variables were in range and checking that certain flags were set before generating an output to the system operator. The alternate routines ranged from re-entry of primary routine, setting a variable or variables to default values through to a less accurate method of the primary routine, for example the alternate routine for range inhibit does not take into account the range at which the target is detected. The overhead in software, caused by the use of a Recovery Block structure depends upon the extensiveness of the acceptance tests and the alternate routines structured within the system software; a typical value resulting from this study was 30 - 40%. The run time overhead depends upon how often the alternative routines are entered, this was found to be in the region of 15 - 20% under the operating conditions of thirty faults (mean) in ten seconds.

The approach of using acceptance tests to flag errors and then using the alternate routines to correct them was avoided, as this quickly leads to a large collection of flags which have to be set, reset and read. This could result in a situation where no error flags are set or reset.

The approach taken was to use the acceptance test to flag an error in the corresponding primary routine as complete failure of the routine and initiate transfer of control to an alternate routine. The assumption made was that if an error was found by the acceptance test then all values generated by the appropriate primary routine were judged to be in error and were regenerated or set to a predefined value.

7.2.1. Integrity of Data.

In order to maintain the integrity of the data base, each variable is only updated in memory after it has been confirmed to be correct. At the beginning of each process the required variables are read into the CPU registers after which the CPU performs the particular process. Only after the acceptance test passes are the updated variables written into memory. This is for two reasons; first, register transfers were considered to be more reliable than memory to register transfers during the process and so were kept to a minimum. Secondly, if the acceptance test fails, a correct copy of the variables is available in memory. The concept of using registers during the process rather than reading the variables from memory is not such a constraint as one might first think as many assemblers allow the user to give registers labels at assembly time. Thus during the process the variable can be given a meaningful label rather than say R6.

7.2.2. Design Discipline.

The immediate implications of the Recovery Block technique imposes an additional element of discipline upon the designer in that he has to divide the total system task into subtasks each of which has an identifiable function which is amenable to acceptance testing. This forces him to think about the total system design and by virtue of packaging into subtasks introduces some element of structure into the program. The subtasks are each associated with a block of code corresponding to a Recovery Block of the form shown in Fig.2.3. These can be linked together to perform a complete software task in a three level system shown in Fig.7.1.
The first level is task direction and points to tasks to be performed in their proper sequence. Level 2 has the format of the Recovery Block for each task, and level 3 contains the coding for each primary alternate and acceptance test routine.

Note that the use of a Recovery Block structure does not preclude the use of defensive programming techniques, often known as Exception Handling[2].

7.2.3. System Performance.

Using the above three level structure a total of 14 runs were carried out, each of 300 system cycles. Of these, five runs were successfully completed with a further one run failing safe during the ten seconds.
This still left a total of eight runs which failed to complete due mainly to the processor trampling through memory.

As previously stated five runs successfully completed the ten seconds, all of these had at least one entry into an alternate routine which prevented the propagation of the original fault. A further two runs had an entry into an alternate routine during the ten seconds but 'crashed' before the ten seconds was completed.

The Recovery Block was found to be capable of coping with data type faults where corruption of data occurs but incapable of dealing with system crashes which may occur due to execution of unimplemented opcode, or execution of unidentified instruction (where operation is uncertain) or by trampling through memory.

7.3. Addition of Watchdog Timer.

The use of the basic Recovery Block as used in the previous section often led to a total loss of function. This loss of function was not flagged by the acceptance test as the test was often not entered under fault conditions. To overcome this the simple expedient of a hardware timer was introduced. On entry to level 2 in the software structure, the timer in the form of a free running counter is loaded with a process time number which is directly proportional to the expected completion time of the process. The task is then initiated, a successful exit from the primary routine leading to a reset of the counter. If the primary routine does not exit in a predetermined time (i.e. the value loaded at the beginning of the process) then the counter goes through zero and triggers a system interrupt, this concept is shown schematically in Fig.7.2.

7.3.1. Recovery Using a Watchdog Timer.

Although it is fairly easy to time out a process due to a failure, the next problem is to return the system to either the alternate routine of the same process or initiate a safe shutdown of the system. Once the interrupt routine has been entered it is not possible to use the program counter contents immediately before the interrupt as a guide to the last segment being processed.

In order for the interrupt service routine to determine the interrupted process, the appropriate process number is loaded into a RAM location at the beginning of each primary and alternate routine. A lookup table can then be used to determine the setting of the program counter which is then loaded to transfer to alternate or fail safe routine, depending upon which routine (primary or alternate) was being processed.

It is possible that if a fault occurs then the task number may be corrupted. To overcome this, the integrity of the recovery mechanism was improved by use of a simple check on the process number to determine whether it is within a predetermined range. If it is found to be out of range then a fail safe routine can be entered. A flow chart of the recovery interrupt service routine is shown in Fig.7.3.

7.3.2. System Performance.

The target tracking system, with the watchdog timer was run for a total of fourteen times. Of the fourteen runs, nine successfully completed the 300 system cycles with valid outputs, with a further three runs failing safe during this period. This left two runs which failed to complete with valid outputs, due either to a system crash or data corruption.

The proportion of runs that completed ten seconds was significantly improved over the basic Recovery Block. This was due to the system recognising that under fault conditions some processes failed to complete within a predetermined time limit. An analysis of the two runs that failed to finish shows that the first would have been able to recover from a particular fault if the unimplemented instruction trap had been used.

The second run failed to finish as the timer had not been started when the fault occurred and the fault led the program counter to be set into memory that was not present, and thus recovery never occurred.

7.3.3. Summary.

The results obtained for the watchdog timer are encouraging when compared with the strategies so far examined, as summarised below.

(a) No Recovery System - 5% of runs successfully completed.

(b) Basic Block Recovery - 35% of runs successfully completed
7% of runs failed safe

(c) Block Recovery with Watchdog Timer
64% of runs successfully completed
21% of runs failed safe.

7.4. Typical Fault Effects.

As a result of injecting hundreds of faults on a microprocessor system, there emerged a number of different fault effects. Before proceeding to further protection and recovery mechanisms, these fault effects are briefly discussed below:

(a) Execution of Wrong Instruction.

An address fault led to execution of instruction at location other than program counter. Following this instruction the program counter was set to a non-instruction word boundary. When fault removed next instruction was an address which corresponded to an instruction for a software interrupt.

(b) Condition Code Error.

A data fault led to a conditional jump based on the wrong condition code.

(c) Opcode Error.

A data fault completely changed meaning of instruction. Instead of loading a register from memory, a different memory location was cleared.

(d) Offset Error in Jump.

A data fault led to a relative jump made to wrong address due to incorrect reading of offset in instruction.

(e) Execution of Wrong Instruction.

An address fault led to execution of instruction at location other than program counter. This instruction (actually an address) led to a reloading of program counter and status register, leading to system crash.

(f) Recovery Block Error.

With the basic Recovery Block, a fault at the end of the primary routine caused the processor to miss the return from subroutine instruction. The processor carried out through the acceptance test, following the primary routine, until it encountered a return instruction which caused a return to the instruction following the call to primary routine which was call acceptance test. This acceptance test actually performed twice on primary routine outputs.

(g) Memory Read Error.

A fault occurred during reading of variables from memory into registers; the registers were left as they were from previous process. Acceptance test failed and recovery occurred by alternate routine which correctly read variables, performed process and passed acceptance test.

(h) Acceptance Test Failure.

After returning successfully from acceptance test, a fault occurred when acceptance test error flag was being checked, the program counter was updated and entry into alternate routine occurred.

(i) Execution of Wrong Instruction.

A fault led to execution of wrong instruction, the program counter was set to a non instruction word boundary and the next instruction was an unimplemented instruction. A trap occurred whose vector had not been set and a system crash followed.

(j) Acceptance Test Failure.

A fault occurred within the acceptance test which led to its failure on good data.

(k) Memory Read Error.

A register was loaded from an incorrect memory address due to a data fault.

(l) Execution of Wrong Instructions.

A register was loaded from memory at wrong point in program due to execution of wrong instruction due to address fault.

(m) Program Corruption.

A corruption of a program location led to an unimplemented instruction trap.

(n) System Data Corruption.

A safe shutdown on the system occurred when primary and alternate routines both failed acceptance test. During execution of primary routine a variable was corrupted in memory and incorrect outputs were given. The alternate routine was to perform primary routine again when incorrect results were also given. This highlights the care necessary when using a repeat of primary routine as the alternate routine.

(o) Subroutine Call Error

A situation occurred where timer was of no use for recovery. A fault occurred at level 2, i.e. CALL PRIMARY, instead of a primary routine being called a subroutine was called whose address was in memory which was not implemented. The timer was set running as primary routine was never entered.

(p) Execution of Wrong Instructions

A fault occurred at the end of a primary routine on instructions to reset timer, time out occurred soon after and alternate routine was successfully entered.

(q) Timer Reset Fault

As a precaution to the above effect, the timer was reset at the beginning and end of every primary routine. A situation arose where, due to a fault the timer was not reset at the beginning of a primary although the task time was loaded and the timer set running. However, there was no ill effect of the missed timer reset as it would have been reset at the end of the preceding process.

7.5. Further Additions to Recovery.

The use of the watchdog timer provided system recovery in 85% of the runs carried out. In order to improve system recovery coverage it is necessary to look at additional facilities which are discussed in the following paragraphs.

7.5.1. Use of Unimplemented Instruction Trap.

One of the runs with the watchdog timer, as summarised in Section 7.3.3. showed that the unimplemented instruction trap can be used for recovery purposes. The Z8000 has a built in unimplemented instruction trap and this can be used for recovery if the vector is set equal to that of the interrupt for the hardware timer. Thus if either an interrupt due to timeout or an unimplemented instruction trap occurs then the same recovery mechanism will be used, as previously described. The run which failed due to this facility not being used was carried out again with its vector set and with the same faults injected, resulting in a successful completion.

Only some microprocessors have this built in facility for detecting unimplemented instructions, though this facility can usually be added by the addition of external hardware. For example, the Texas 9900 which has about 2% of its opcode field as unimplemented can use external hardware as given in Ref.25.

In addition most processors have a software interrupt facility. If this is not required by the system software then the software interrupt vector should be set equal to that of the hardware timer, so that an unexpected software interrupt due to a fault will not cause a system crash.

7.5.2. Default Data Bus.

In most real time systems there are areas of the memory map that are not filled by memory devices. Thus if the program counter is inadvertantly set somewhere within this unimplemented area then during a read operation the data bus will be floating. This can be made use of by attaching resistors onto the data bus, in the form of pull up and pull down resistors, so that the data bus defaults to an instruction such as software interrupt when not driven. This is shown by example in Fig.7.4. for a 4 bit data bus where a software interrupt is represented by 1100 (binary). The resistors should be of sufficiently high value in order to prevent excessive current drain. During normal operation the bus will be driven high and low as required by CPU and memory devices. The first instruction that the processor executes after jumping into unimplemented memory is a software interrupt whose vector is set equal to that of the timer and thus initiates a recovery. This mechanism provides an earlier fault indication than the watchdog timer in the situation of the processor jumping into unimplemented area, although the timer is still invaluable for recovery if the processor executes an incorrect section of code. In addition this mechanism provides a fault indication if the processor jumps into an unimplemented area of memory before the timer is started.

7.5.3. Trap Area.

It was found that on many occasions, due to faults, that the return statement at the end of a process was missed and the processor continued into the next section of code. In this situation recovery would still take place by watchdog timer or by execution of an unimplemented instruction. However to speed recovery and to reduce still further any inadvertant action, a trap area can be used after each return statement, i.e. between each process. This trap area, shown schematically in Fig.7.5., would consist of a gap equal to the maximum length opcode in words of the processor. This trap area would consist of software interrupt instructions whose vector was set equal to the timer recovery procedure. Thus if a return was missed due to a fault then a software interrupt would occur and recovery take place.

7.5.4. Performance Counter.

It has been shown that with certain additions to the Recovery Block, it is possible in a single microprocessor system to recover from all, as far as can be seen, transient hardware faults. However, in many real time systems it is not sufficient to use an alternate routine cycle after cycle in the case of a prolonged fault or software design error as degraded performance may only be acceptable for a limited period of time before a different system strategy is required. This is almost certainly true in a situation with recursive calculation where an alternate routine may use last value or a default value. Thus it is suggested here that in many applications a counter be used within the alternate routine to count consecutive or total entries into the routine. If the count is exceeded then another alternate, for example use of another sensor, or fail safe routine can be entered.

7.6. Extensions to the Recovery Block

This section summarises the possible extensions made by the author, the majority of which have been implemented.

(a) An unimplemented instruction trap can be used to speed response to faults. This trap is internally implemented on processors such as the Z8000, 68000 and can be readily implemented in hardware on others such as the TMS 9900.

(b) A timer can be used to ensure that primary and alternate routines do not take longer than expected to execute or finish before a minimum time.

(c) A set number of automatic retries can be used before classifying the fault as permanent or transient.

(d) The use of pull up and pull down resistors to provide recovery when program counter is set into an unimplemented memory area.

(e) For system critical variables, it may be re cessary to keep a copy in both memory and allocated register within CPU. This can obviously only be used for one or two variables.

(f) In some instances it may not be possible to perform the acceptance test on the alternate routine due to time constraints.

(g) In some instances it is not advisable to carry out an acceptance test on the alternate routine, if the size of the routine is less than that of the acceptance test.

(h) A count may be necessary within the alternate routine as degraded system performance may only be acceptable for a limited period of time.

(i) A trap area can be used between processes to eliminate the possibility of inadvertantly going from one process to another through omission of a return.

A more generalised form of the Recovery Block can be found in Fig.7.6. which covers some of the points mentioned above, which are not covered by the basic Recovery Block. In this figure block A is general, the output of A may be to P or Q or output or other alternates.

Chapter 8. Single Microprocessor Study Conclusions.

The single microprocessor study has shown that by observing certain formats for software layout, most transient hardware bus faults are recoverable. This recovery strategy produces a small overhead in running time and memory. The Recovery Block technique used also enforces a degree of design discipline onto the software engineer to produce a structured format to his software.

8.1. Acceptance Test.

It has been found that care must be taken in designing the acceptance test for a particular process. A compromise must therefore be made between the amount of testing in the acceptance test, and the overhead incurred.

8.2. CPU Local Storage.

The Recovery Block, in its simplest form provides protection and recovery mainly from faults that lead to data corruption. The integrity of the data is improved by a procedure where variables are read into CPU registers, followed by the particular process, and finally the updating of variables only after the acceptance test has been passed. This method leads to a greater probability that data within memory is uncorrupted, and is already available in some high level language compilers.

8.3. The Watchdog Timer.

The introduction of a watchdog timer resulted in a small software overhead, additional software was used for setting up, starting and resetting the timer. The overhead was less than one per cent for software, in addition to a simple counter for tuning system operation. The use of a watchdog timer highlights the importance of a system approach to fault tolerance through the combined use of hardware and software to increase the availability of the system. The increase in availability that was obtained by the use of a Recovery Block structure and a watchdog timer is shown in Fig.8.1.

The recovery mechanism used consisted of entry to an alternate routine either by failure of the relevant acceptance test or following execution of a fault detection interrupt service routine. A simple interrupt service routine kept recovery time to a minimum. The number of the process being executed at the time of the fault was read, and a check was made that it was within an expected range. The process number was then used as an entry to a process re-entry look up table stored in ROM, followed by a jump to the relevant process re-entry point. System variables are only updated following successful completion of the relevant acceptance test; it is assumed that a copy of valid system variables remains within the RAM area. If this assumption is invalidated, for example by a momentary power failure, then a fail safe state is entered shortly afterwards through the mechanism of a count being exceeded within an alternate routine.

8.4. Default Data Bus.

The use of a watchdog timer generally provides recovery when the microprocessor's program counter is corrupted to a value outside the segment being processed. In addition, there are situations when the program counter stays within the segment, but the segment is either completed too quickly or not within time; this latter case is particularly important in real time systems. If the program counter is corrupted to a value outside of the segment being processed, then it can be situated in one of two areas of memory. First, the program counter can be corrupted to a value which corresponds to another segment, and secondly the program counter can be set to a value which corresponds to unimplemented memory. This latter situation arose in the study and recovery time was decreased by the use of default resistors on the data bus. These resistors were used to trigger a software interrupt when the microprocessor attempted to execute an instruction from unimplemented memory. Furthermore the same recovery routine can be used as that for the watchdog timer.

8.5. Microprocessor Dependent Facilities.

A growing number of microprocessors have traps for detection of illegal conditions such as attempted execution of illegal instruction, bus error and division by zero. System recovery can take place under these conditions if the trap vectors are set equal to the vector of the hardware timer.

8.6. Use of Trap Areas.

The majority of real time systems have critical areas of software where a correct procedure must be carried out before an action can be taken. The Recovery Block technique is useful in this situation whereby the setting of flags can be checked within an acceptance test. However, this situation can be improved by the use of trap areas between segments in a critical area of software. This prevents the microprocessor from running on from one segment into another. Recovery takes place if the program counter is set equal to an address within the trap area, provided that the trap is filled with a suitable software interrupt.

8.7. Performance Counter.

In real time control systems it is important that a counter is provided within alternate routines as degraded performance may be acceptable only for a certain period before a different system strategy is required.

8.8. Built in Test.

A built in test facility is often used for operator confidence and for diagnosing faults in the field. The Recovery Block technique can be used as an aid in testing and diagnosing faults. It was previously mentioned that a counter can be used within certain alternate routines so that continued degraded performance is prevented. Whilst the system is in a standby state, the counts from the alternate routines can be used as an input to the built in test equipment and provide information on possible faults. For example, a certain alternate routine entry may be associated with the defective reception of information from a peripheral; this information can aid test equipment in diagnosing a fault.

8.9. In Conclusion.

The use of a Recovery Block structure established that the mechanism is a useful tool which can be integrated into the design of real time system for improved availability. The most important additions to the basic structure are the use of a watchdog timer and a simple counter within alternate routines. It has been stated that the Recovery Block is not capable of recovering from software errors due to incomplete or inconsistent requirements specification. This situation can be improved by the use of independent design of alternate routines to simulate an N-Version Programming approach without the need for massive redundancy, although this is very difficult to achieve in practice.

Chapter 9. Introduction to the Distributed Processing System.

The single processor study demonstrated that increased availability under prescribed fault conditions was obtained using protective redundancy. This confirmed that the propagation of faults from one process to another could be stemmed by the use of a fault detection and recovery strategy. The next objective of the research study was to investigate the possibility of increased availability for a distributed processing system undertaking the tasks of target tracking and missile guidance, as described in Chapter 4. The nature of the increased system complexity required to undertake these tasks, together with the locations in which they would normally be undertaken involved the decomposition of the system into subsystems.

The first objective was to establish a design philosophy for communication between the subsystems; this being described in the following section.

9.1. Design Philosophy for Inter Processor Communication.

The Recovery Block technique ensures that only valid data is passed from one process to the next, by use of the acceptance test. The following process simply takes the data and uses it without any need for testing its validity. This approach can then be extended to a distributed processing environment in the following manner.

Consider the transmission of data from one microprocessor subsystem to another using a communications link. The use of a Recovery Block structure within each subsystem ensures that only valid data is transmitted. The design philosophy for message passing follows implicitly, i.e. that data testing is carried out at the point of maximum information (transmission) with the absence of testing data on reception. This is shown schematically in Fig.9.1. The testing of data is carried out by an acceptance test prior to transmission, the data is assumed to be valid if it is received correctly with respect to the particular communications protocol. If a transmission failure occurs, for example incorrect parity, then a request for re-transmission can be made.

When message passing is carried out between subsystems then the transmitting subsystem is said to be active whilst the receiving subsystem is passive. The transmitting subsystem has a responsibility to provide valid data with the use of local recovery if necessary, while the receiving subsystem need only wait for data.

9.2. Local Recovery Strategy.

The initial aim for recovery from a fault within a distributed processing system is the attempted recovery on a local basis, that is within the subsystem. A schematic diagram of local recovery is shown in Fig.9.2. This figure shows the importance of localising the effect of a fault and the prevention of propagation to other subsystems.

In view of the experience and results obtained for the single processor study, it was decided to continue with a similar strategy for each of the microprocessors within the distributed system. The Recovery Block structure was discussed in Chapter 7 and when implemented within each microprocessor subsystem provide the basis for local recovery.

The absence of reception of expected data leads to another principle, i.e. the message transfer proceeds only in one direction. If a message fails to arrive then the receiving subsystem must not attempt to diagnose the failure to transmit, instead it must initiate global recovery after a predetermined time period.

The concept of global recovery is introduced in the following section.

9.3. Global Recovery Strategy.

In a real time distributed processing system, it is possible that local recovery may fail or that communication between processors may fail. Under these circumstances, in a master/slave system then the master can wait only for a predetermined time before action has to be taken. This action of global recovery can take place in the event of failing to receive data from a slave. Global recovery can simply be seen as failing to pass the acceptance test of a routine in the master which is requesting data, and the subsequent transfer to an alternate routine.

The above strategy is illustrated in Fig.9.3., which is described below:

Cycle Time	Master	Slave	Remarks
1	Request for data	Satisfactory response	(a) Acceptance test in master passes: no communication faults.
2	Request for data	Satisfactory response	(a) Acceptance test in master passes. (b) Fault occurs in slave after transfer of data. (c) Local Recovery attempted in slave but faiks.
3	Request for data	Unable to send data	(a) Master requests data, four retries are carried out. (b) Acceptance test in master fails as no data available. (c) Alternate routine in master entered.
4	Request for data	Satisfactory response (slave sub-system able to transmit valid data)	(a) Acceptance test in master passes.

This alternate routine then provides data that can be used by the system for continued operation. This data may be a default value or the last correct value received. The action of transfer to an alternate routine prevents the maximum system latency being exceeded. However, continued entry of this routine may be dangerous to the system and may occur in the presence of a permanent failure of one of the microprocessor subsystems. This type of failure is considered in the following section.

9.4. Task Swapping.

If a permanent failure occurs in a slave then global recovery is not possible over a prolonged period, due to the repeated entry of an alternate routine within the master. Such a failure would only be retrieved if redundancy were to be included. Under these conditions it becomes necessary to use an alternative processor to carry out the function of the failed slave.

Having described a recovery strategy for the distributed processing system, the next point for consideration is the manner in which the system is distributed. The criteria governing this,together with the approach which was adopted is discussed in the following section.

9,5. Functional Decomposition of System.

The manner in which the functional decomposition is carried out is an essential feature of the system recovery strategy[26]. The factors to be considered in this respect being as follows:

1. Inter processor communications to be kept to a minimum.
2. Separation by function or process
3. Considerations of physical locality of functions.

The function of the distributed processing system was to perform the target tracking process and the missile guidance loop equations, which are divided into those of the digital controller and the missile autopilot. This provided a natural split into three sub-functions, each of which could be perform.ed by a separate microprocessor. This natural division also meets the three criteria stated above which is shown schematically in Fig.9.4., with the realisation of the sub-functions being described in Chapter 5. The generation of raw target data is carried out by a Fortran program running on the PDP 11 in a similar manner to that used in the single processor study. Intercommunication between the subsystems was carried out using a high integrity data highway link. For continuity of design and use of existing software, the Z8000 microprocessor was chosen as the processing element for each of the subsystems.

In order to effectively monitor the detection and recovery from faults, it was necessary to inject faults onto the distributed processing system. A description of how this was achieved is given in the following section.

9.6. Injection of Faults in Real Time.

In the single processor study faults were injected by halting the processor, selecting the fault by switches and then single stepping, one system instruction being executed with a corrupt address or data bus. This approach, when extended into a distributed processing system would require the synchronisation of all the processors, which was considered to be an over complicated solution. The problem then was how to inject faults in real time on one of the microprocessor subsystems.

Initially, faults were to be injected by means of pseudorandom generators[27]. This was dropped in favour of the following approach as it was considered that it would be more informative by injecting repeatable faults in known positions of the software, in so far that the type of fault injected is directly correlated to the observed failure at a systems level.

9.6.1. Mechanism of Fault Injection.

The mechanism of fault injection used in the distributing processing research was as follows. A hardware register is loaded by the microprocessor with a fault address, prior to the operation of the system. When the processor reaches this address in the software, a comparator is activated by the two addresses (i.e. hardware register and address bus) being identical. This is shown schematically in Fig.9.5. A non-maskable interrupt is then generated and the interrupt service routine activates the fault. For example, the interrupt service routine may either read a variable and corrupt it, or corrupt the stack or stack pointer.

At the end of the short interrupt service routine the microprocessor loads the hardware register with the next fault address. Finally the 'return from interrupt' instruction returns control to the module being executed prior to the interrupt, or to another address if the stack has been corrupted. Provided that the interrupt service routine is short enough, say less than 1% of a system cycle, then a fault can be injected in real time.

9.6.2. Specific Cycle Fault Injection.

This mechanism can be used to inject a fault within a specific predetermined cycle as shown in Fig.9.6. The interrupt service routine then reads the cycle number; if the cycle number is the one in which the fault is to be injected then the predetermined fault is allowed to occur. On every other cycle, the cycle number is found not to be equal to the required cycle number and a 'return from interrupt' instruction is then executed. The overhead incurred in adopting this procedure was in the order of a few tens ot micoseconds which was generally short enough not to invalidate the system operation for the research model.

An alternative method of injecting a fault onto a specific system cycle would be the use ot a maskable interrupt which could then be enabled on the specific cycle. This approach was not used as it involved modifying the system software, that is, it requires the addition of enable and disable interrupt instructions and a recompilation of software if a different fault address is required.

Chapter 10. The Distributed Processing System Description.

As described previously the distributed processing system used in this study is based on three active processor subsystems which perform the system function, together with a standby processor subsystem for failure recovery viz. task swapping as shown in Fig.10.1. Communication between subsystems was carried out using a high integrity serial data highway, a description of which is contained in this chapter.
The microprocessor used in the single processor study was utilised as the basis for one of the subsystems. The other three subsystems consisted of identical processor cards which were constructed to the author's design. The facilities offered by these common processor cards are described in Chapter10.1. The link selector shown also in Fig.10.1. comprises a manual switch arrangement for routing the program loading of the subsystems via the RS 232 data link.

The requirement for the data highway between the microprocessor subsystem was based on the following criteria:

1. Distributed processing power
2. High communication bit rate
3. Ability for system expansion
4. High integrity communications

It was considered important to make a choice of data communication system which had an established message format and protocol. This led to the decision to implement MIL - STD 1553B[28], which has been developed for high integrity data communications between aircraft subsystems.
An overview of MIL - STD 1553B can be found in Appendix F.

10.1. Central Processing Unit.

The subsystem processor card designed for the real time control system is based around the Z8000 microprocessor. An RS232 serial interface is included on the card to provide communication with a visual display unit. The default baud rate was set to 9600, but different rates can be selected by the interconnection of wire wrap pins on the card.

Details of the baud rate selection can be found in Table 10.1.

The card contains 4K bytes of static RAM and allows for up to 8K bytes of EPROM. A 4K byte monitor on the card is derived from that on the Am96/4016 Evaluation Card.[24] The memory maps of each processor subsystem are identical and are as shown in Fig.6.3. The card also contains the logi c, as described in Chapter 9.6. for the injection of faults in real time.

The circuit diagram, layout diagram and parts list are shown in Figs. 10.2. and 10.3. and Table 10.2.

10.2. Microprocessor to 1553B Interface.

The data highway interface was designed to meet the requirements of MIL - STD 1553B for communication with a Z8000 microprocessor. The interface was capable of acting as either a bus controller or as a remote terminal. The position of a dual-in-line switch on one of the interface cards decided which mode of operation was to be used for the terminal. The design uses a single twisted pair bus, although the standard allows up to three redundant buses, in addition to the active bus. The interface appears to the Z8000 as a number of memory addresses as shown in Table 10.3.

A schematic of the microprocessor to 1553B interface can be found in Fig.10.4. and shows that the message path between the serial bus and the Z8000 is achieved by the use of a 32 word FIFO. A control register decides whether the word to be sent or received is a command, data or status word. The interface was designed on the principle that a remote terminal is always ready to receive a message but is not always ready to send a message.

A simple time out circuit on the transmitter of the interface precludes continuous transmission longer than 800 microseconds, implemented as a monostable which is triggered by a request to send a message. Thus the failure of a bus controller results in a quiet bus with no transmissions, due to the time out. The 1553B standard allows ten message formats although only two of these are required for this study, these being bus controller to remote terminal transfer and remote terminal to bus controller transfer.

10.2.1. Control and Status Register.

Control and status information within the interface consists of two registers, one for read and one for write, having the same address. The function of the control and status register bits is shown below; these bits form the data word which is either read from or written to the status register. Each bit of the status register is valid only when the terminal is either a bus controller (BC) or a remote terminal (RT). The exception is bit 9 in the read status which is valid in both modes of operation.

Read Status:

Bit No.	Title	Function
0 - 7		Not used.
8	ME(RT)	A logical '1' indicates that the last message was invalid.
9	$C/\overline{RT}$	A logical '1' indicates that the terminal is configured as a bus controller. A logical '0' indicates that the terminal is configured as a remote terminal
10	BUSY(BC)	A logical '1' indicates that a busy status return was received from a remote terminal.
11	BUSY(RT)	A logical '1' indicates that the remote terminal is unable to send data.
12		Not used.
13	ME(BC)	A logical '1' indicates that the message error bit was set in the last status return.
14	OR(RT)	A logical '1' indicates that the FIFO contains valid data.
15	$T/\overline{R}$(RT)	A logical '1' indicates that a request for data has arrived. A logical '0' indicates that data has arrived in the interface.

Write Status:

Bit No.	Title	Function
0	BUSY(RT)	A logical '1' sets the busy bit within the status word.
1	DBCA(RT)	A logical '1' sets the dynamic bus control acceptance bit within the status word.
2	SUBFLG(RT)	A logical '1' sets the subsystem flag within the status word.
3	SERREQ(RT)	A logical '1' sets the service request bit within the status word.
4 - 15		Not used.

The interface was built on two Eurocards; the circuit diagrams, layout diagrams and parts list can be found in Figs.10.5., 10.6., 10.7., 10.8. and Tables 10.4. and 10.5. Figure 10.9. shows a photograph of the two interface cards.

The operation of the interface is best described by considering its use as a bus controller and then as a remote terminal under the operations of sending and receiving messages.

10.2.2. Message from Bus Controller.

Consider the interface configured as a bus controller, and requiring to send a message to a remote terminal. Initially the microprocessor loads the FIFO with the message to be sent followed by the loading of the command word with the transmit/receive bit set to receive. Note the transmit/receive bit is set depending upon the direction of the message with relation to the remote terminal being addressed. When the command word has been loaded, the microprocessor then initiates the transfer, as shown in the timing diagram in Fig.10.10. The low to high transition of the initiate command enables the Manchester Bi-Phase encoder, which sets the SEND DATA output high when it is ready to receive data. The command word is converted into serial data, which is clocked into the encoder at a rate of one bit a microsecond. After the sync and encoded data are output, the encoder adds on an additional bit which corresponds to the parity for that word.

The encoder produces bipolar outputs which are used to drive an isolating transformer via a long tailed pair as shown in Fig.10.7.b. The connection between the isolating transformer and the bus is achieved by means of a stub and a coupling transformer as shown in Fig.10.11. The coupling transformer for each interface is housed in a shielded box at the back of the expansion box.

When SEND DATA goes high after transmission of the command word the first data word is clocked out of the FIFO. The data word is converted into serial data and then clocked into the encoder when the encoder is ready to accept data. The converted serial data word is preceded by a data sync which is different from the sync which precedes the command word, as shown in Fig.F.2. After the last word has been transmitted, the bus controller then expects to receive a status word from the addressed terminal to confirm that the message has been received. If this status word is not received within 15 microseconds of the last data word being sent, then a response time out occurs. Note the 1553B standard requires that a bus controller wait at least 14.0 microseconds before allowing a no response time out to occur; no maximum time period is specified within the standard. The time out can be used to inform the microprocessor that message handshaking has failed; which can then be followed by a re-transmission or other predetermined course of action.

10.2.3. Message to Bus Controller.

Consider now the operation of a bus controller requesting a message from a remote terminal. A subaddress field of five bits within the command word can be used to signify, for example, a request for a particular data type. The controller sets the word count field equal to the required message length, the transmit/receive bit equal to transmit, and the address and subaddress fields to their relevant values. This command word is loaded into the command register followed by an initiate transfer command from the microprocessor subsystem. The low to high transition of the initiate command enables the Manchester Bi Phase encoder; the timing diagram is shown in Fig.10.12. The serial form of the command word is clocked into the encoder when it sets SEND DATA high.

The addressed terminal identifies its own address within the command word and signals the subsystem processor that a message is required. If the message has not been preloaded into the interface then the subsystem would have set the busy bit within the status word which is transmitted to the bus controller. The status word is decoded by the Manchester Bi-Phase decoder which sets TAKE DATA high. The bus controller recognises that the remote terminal was unable to transmit the message at that time, it then waits for a predetermined period, before re-transmitting the command under subsystem control. The period of waiting is under control of the subsystem processor, and was typically set between fifty to a hundred microseconds.

During the period ot waiting, the transmitting subsystem processor identifies the relevant message and loads it into the interface. The busy bit in the status register of the remote terminal is also reset so that when the request is received again then the message is automatically transmitted. On this occasion, the bus controller decodes the status word and recognises that the required message follows the status word. The data is loaded, one word at a time into the FIFO; after the last data word CONTIGUITY FAIL goes high since there was no bus activity for a period of four microseconds since the last data word. The length of the message requested is checked with the number of words received to confirm that the message has been correctly received.

10.2.4. Message to Remote Terminal.

Consider the operation of a remote terminal receiving a message. The first word received is the command word which is decoded by the remote terminal. Having ensured that the message has the correct address, the interface loads the word count into a latch and clears the FIFO ready for the message. In addition a signal VALID COMMAND SYNC goes high which starts the receive cycle; the timing diagram is shown in Fig. 10.13.

As each word arrives it is decoded into serial data with the decoder setting a VALID WORD signal high if the word is valid. The serial word is converted into a parallel 16 bit word, loaded into the FIFO and the word counter is incremented. At the end of the message the interface recognises a period without data syncs, and sets CONTINGUITY FAIL high. The value of the word counter is then compared with the word count from the command word. If these two values are equal then the message has been correctly received and the subsystem processor is interrupted to indicate the presence of a message. The Manchester Two Bi-Phase encoder is enabled and the status word is sent to the bus controller. If the word counts are not equal then an error has occurred and the subsystem processor is not interrupted. The occurrence of an error sets the message error bit in the status register and the status word transmission is suppressed.

10.2.5. Message from Remote Terminal.

Consider the operation of a remote terminal sending a message. When a request for data is received, a signal VALID COMMAND SYNC goes high, as shown in the timing diagram in Fig. 10.14. The encoder is enabled and the status word is clocked into the encoder and transmitted. On the falling edge of SEND DATA, the interface determines whether a message has been loaded into the FIFO.

If a message has been loaded then one word is read at a time from the FIFO; each word is converted into serial data before being sent as part of a contiguous message. However, if no message has previously been loaded into the FIFO the busy bit is set within the status word return. This indicates to the bus controller that the remote terminal was unable to send a message in response to the request. Due to the time constraints of the 1553B standard (i.e. respond with status word within 12 microseconds) there is insufficient time to load a message into the FIFO after receiving a transmit command and before it is necessary to send the status word. The subsystem processor is then interrupted and can then load the required message into the FIFO and release the busy within the status register. On the next request to transmit the message is sent to the bus controller.

10.2.6. 1553B Protocol Fault Injection.

The encoding and decoding of Manchester Two Bi-Phase Level data within the interface was carried out by a customised integrated circuit, the Harris 15530.[29] This integrated circuit sets the word length to 20 bits as defined by the 1553B standard. An alternative integrated circuit similar to that above, the Harris 15531[30] was used within one of the interfaces and allows 1553B protocol faults to be injected onto the bus. The integrated circuit is similar to that described above except that the frame length and parity are programmable for both the encoder and the decoder. A frame length of between six and thirty two bit periods can be obtained with this device, which is set up by writing to address 6FEØ. The bit pattern and the corresponding frame length can be found in Table 10.6.

This interface was also constructed on two Eurocards, whose circuit diagrams are found in Figs.10.5. and 10.15. The corresponding layout diagram and parts list for Fig.10.15. are to be found in Fig.10.16. and Table 10.7.

10.3. Communications Software.

The available time for designing and building the 1553B interface was limited, therefore the decision was made to use the Z8000 processor to pass data in and out of the interface rather than use DMA which would have been more elegant. However, this decision did not affect the performance of the distributed processing system as sufficient free time was available to allow the processor to transfer the data.

The communications software written for this study is described by considering the sending and receiving of messages to and from the bus controller and a remote terminal, as follows:

10.3.1. Message from Bus Controller.

The sending of a message is performed as shown in Fig.10.17. The processor clears the FIFO prior to writing the message one 16 bit word at a time into the FIFO. When the message has been loaded, the command word is loaded into the command register. This command word contains the address of the remote terminal which will receive the message and the word count of the message. Finally, a send command is given and the message is sent under control of the interface.

Under normal conditions the message transfer is then complete; however, if no status return is received from the remote terminal in question, then an interrupt is generated and the sequence can be repeated.

10.3.2. Message to Bus Controller.

The request and reception of data is performed as shown in Fig.10.18. The message sequence starts with the processor loading the command word register and then initiating the transmission. If the busy bit is set in the status word from the remote terminal then an interrupt occurs. The interrupt service routine increments the busy count (number of requests given a busy reply), clears the interrupt flip flop and returns to the calling program which repeats the sequence. If the busy is not set in the status word then no interrupt occurs and the message is read from the FIFO within the interface after a short delay.

10.3.3. Message to Remote Terminal.

The reception of a message is performed as shown in Fig.10.19. When data is expected from the controller the interrupt is enabled. On reception of a receive command, an interrupt is generated and the message is loaded into memory. On return from interrupt the remote terminal then disables the interrupt.

10.3.4. Message from Remote Terminal.

The sending of a message by a remote terminal is performed as shown in Fig.10.20. It is assumed that the busy bit within the status register is set; when a request for data first appears the busy reply is given. The request for data triggers an interrupt; the interrupt service routine then loads the message into the interface. When the request appears again the message is sent, this condition is recognised by the subsystem processor which then sets busy for the next request.

10.4. Systems Integration and Test.

As for the single processor case, system integration and test programs were developed for this phase. These consisted partly of programs written for the single processor togehter with communications test schedules. These programs have not been included in this thesis.

Chapter 11. Design Strategies: Distributed System.

This chapter presents strategies for detection and recovery from transient and permanent hardware faults,and their implementation in hardware and software within a real time distributed processing system. The approach was, first to inject faults onto the control system which had no recovery mechanism. Having gained experience from the single processor study on the effect of faults, it was felt unnecessary to inject a large number of random faults but instead to inject faults to give typical or specific faults. Having obtained a baseline, the basic Recovery Block was implemented upon the target tracking and digital controller software.

Other techniques, for example the use of a watchdog timer, developed in the single processor study were then implemented in order to localise the effect of faults. Global recovery was used to prevent a system crash or an unsafe system state when the localisation of the effect of faults was not possible.

The performance of the distributed processing system was obtained using the results of tracking a single target, whose characteristics are described in the following section. The subsystem is operated wholly in track mode and for the purposes of the distributed system, a run is considered to start at missile launch.

11.1. Target Characteristics.

The target used for the distributed processing study was different from that of the single processor study and had the following characteristics:

START POSITION	4000	4700	200	(metres)
HEADING	- 100	4000	200	(metres)
VELOCITY	250			(metres/second)

This target was chosen as it gave a missile angle characteristic, as shown in Fig.11.1., which has two phases of missile flight, i.e. that of gathering and the terminal phase. In addition, the missile range is not equal to the target range until approximately 10.8 seconds, as shown in Fig.11.2., thus allowing the system to recover under difficult fault conditions.

The target tracking software was modified slightly from that used in the single processor study, and involved the use of 120 sectors to represent 360° instead of the 30 sectors previously used. This increase in the number of sectors allows more accuracy to be obtained in target tracking, due to the higher resolution.

The use of 120 sectors gives a sector spacing of 3°, and the effect of this can be seen in Fig.11.1. The missile does not lie on the exact angle as the target during the terminal phase but can still said to be tracking the target. Tracking can be justified as the missile lies within the same 3° sector as the target, and the system cannot distinguish one edge of this sector from the other edge. Thus during the terminal phase the missile believes it is on the same azimuth as the target, and a target hit is considered to have occurred if the missile angle is within the same 3° sector when the ranges are equal. This situation is adequate for the purposes of demonstrating system recovery, but can be improved by the use of smaller sectors and the use of feedforward terms in the missile guidance loop.

The operation time of a single run was extended from ten to fifteen seconds, this was simply a convenient time which was greater than the time for the missile range to be equal to the target range. In taking results the criterion taken was to compare the missile angle under fault conditions with the true missile angle obtained under no fault conditions. Each run was continued to fifteen seconds even if a target hit occurred before this time. The measurement of performance is described in the following section.

11.2. Performance Index.

A quantitative measure of performance was required to assess the performance of the system under different fault conditions. The use of availability as a measure is quite good but does not differentiate between a single long unoperational period and many short periods. In many applications, it is not sufficient just to recover from a fault but it is important that fast recovery takes place as in the case of a missile tracking a target. In addition, the time at which a fault occurs is important, for example, a fault occurring at nine seconds after missile launch has a higher probability of disrupting system performance than a fault occurring at three seconds.

The missile flight consisted of two distinct phases, that is the gathering and the terminal phases. During the gathering phase the missile to target angles are large in contrast to the small angles obtained during the terminal phase. Since the guidance control is closed loop, the system recovers naturally from propagated data corruption type faults. However, the natural recovery period is likely to be significant and may result in a failure of the mission particularly if the fault occurs during the terminal phase. Thus it is important that data corruption type faults are not allowed to propagate and that the system is always in a known state.

In order to penalise slow recovery and large errors from the expected performance, the following measure, called a Performance Index was used

$$\text{Performance Index} = \int_0^A t(\text{error})^2 \, dt \qquad .....(11.1.)$$

The upper time limit of the integral occurs when missile range is equal to target range.

11.3. System with No Recovery.

Initially the system was configured as shown in Fig.11.3. without any protection or recovery schemes to provide a baseline set of results. Two types of faults were considered, that of data corruption and faults that caused the digital controller to crash.

11.3.1. Data Corruption Type Faults.

Using the mechanism described in Chapter 9.6. faults were initially injected to produce data corruption effects. First, consider faults introduced during the gathering phase, i.e. up to about eight seconds after the start of the run. The effect of corrupting the target angle presented to the missile guidance loop can typically be as shown in Figs.11.4. and 11.5. Fig.11.4. shows the effect of corrupting the target angle to a value of - 3° for a period of eight iterations (1/15th second) at two seconds after the start of the run. This value is a legal target angle, however such a jump in target angle is unlikely to occur under no fault conditions. This results in a maximum deviation of 6.05° and a performance index of 65.0(seconds, degrees)^2. Fig.11.5. shows the effect of corrupting the target angle to a value of - 6° for the same period at four seconds after the start of the run. This also gives a maximum deviation of 6.05° with a performance index of 112.3(seconds, degrees)^2.

A data corruption type fault occurring in the output of the digital controller corresponds to the missile being given an incorrect guidance demand. The effect of setting the guidance demand to zero at 1/4 second from the start for eight iterations is shown in Fig.11.6. This figure shows a maximum deviation of 8.49° and represents a performance index of 152.2(seconds, degrees)^2. The effect of data corruption occurring during the gathering phase, as shown in Figs 11.4., 11.5. and 11.6. is to change the plot of missile angle but does not affect the terminal phase of the missile.

The time taken to recover from a data corruption fault within the gathering phase was between two and six seconds. If this recovery period is repeated during the terminal phase then the effect of the fault is to cause the missile to miss the target. In the terminal phase the recovery period was generally shorter as shown in Figs. 11.7. and 11.8. which indicates that tracking was lost for between one and three seconds.
The effect of an uncontrolled overflow, due to a large target angle, in the controller's calculation of lateral acceleration is shown in Fig.11.9. This effect is quite severe causing the missile to slew rapidly, giving a maximum deviation of 14.0° with a corresponding performance index of 14195.8(seconds, degrees)^2. Tracking is regained three seconds after the fault was introduced during which time the target was missed. The overall effect of data corruption in the terminal phase in a system without

recovery is that there is a high probability that the target will be missed.

11.3.2. Controller Crashes.

The next stage was to consider the type of fault that led to a controller crash, i.e. a total loss of system function. Typical causes of system crashes were found by studying the single processor results; a list of these causes can be found in Table 11.1.

Twelve runs of the system were carried out, each run was faulted by one of the fault types listed in Table 11.1. The faults were injected within the calculation of the difference equations by substituting one of the instructions in Table 11.1. for a system instruction. Of these twelve faults, all caused a loss of system function except fault type number 2. The introduction of a relative jump meant that the program counter stayed local to the correct value and a system crash did not occur; the effect was one of data corruption. This cause was eliminated from further consideration of faults that cause the system to crash if no protection or recovery is applied.

11.4. Basic Recovery Block.

The previous section identified two different types of fault and their effects; the next step was to implement the basic Recovery Block and monitor its effectiveness in a distributed processing environment under these fault conditions. The basic Recovery Block was implemented within the target tracking processor and the digital controller processor to localise the effect of faults on total system performance. The implementation is described below followed by the resulting effect of the faults.

11.4.1. Target Tracking Processor.

The basic Recovery Block implementation used was the same as for the single processor study (see Chapter 7.2.) except that the software was modified to allow 120 sectors per revolution.

11.4.2. Digital Controller Processor.

The digital controller, as described in Chapter 5.3. consisted of the addition of four difference equations. Each of the five units (four difference

equations plus the addition) had its own Recovery Block with the acceptance test defined as ensuring the output is within the worst case limits.
The estimation of worst case limits can be found in Appendix G.
The outputs of the four parallel units and their addition can be found in Figs. 11.10. and 11.11. and satisfy the results obtained in Appendix G.

In addition to acceptance testing, any overflow following an arithmetic operation resulted in the entry of the appropriate alternate routine. For simplicity, the alternate routine was to re-execute the primary routine.

11.4.3. Data Corruption Faults.

The data corruption faults as described in Chapter 11.3.1. were introduced into the system with the basic Recovery Block. Of the faults introduced into the target tracking processor all were captured by the relevant acceptance tests. This resulted in no degradation in the plot of missile angle, even though a default or last value was used on several occasions. The explanation for this is that the output of the target tracking process is slow moving, with the target azimuth being updated once per second.

Now consider faults injected into the controller software, as before, the output of the digital controller difference equations was corrupted and set to zero. The acceptance test was entered and the output passed the test. The resulting missile angle plot was the same as for the system with no protection, i.e. as in Fig.11.6. However, the effect of this fault occurring during the gathering phase does not influence the system's ability to enter the terminal phase.

The effect of allowing a large transitory target azimuth appear as input to the missile guidance loop was shown in Fig.11.9. This caused overflow in the digital controller's difference equations. However, with the basic Recovery Block implemented within the target tracking processor, then the acceptance test trapped the large swing away from the target being tracked. The alternate routine was then entered and the previous value used; this resulted in the plot of the missile angle being equal to that under no fault conditions. Thus, the extent of the fault was localised within the target processor and was not allowed to propagate to the digital controller.

11.4.4. Controller Crashes.

The causes ot system crashes, as listed in Table 11.1. except fault type number 2, were introduced into the controller software with a basic Recovery Block structure. All the runs failed to complete i.e. a system crash occurred, except number ten (POP instruction). This was due to the structure of the Recovery Block. The POP instruction results in the correct return address of a subroutine being taken off the stack, this led to the processor pointing to the wrong calling address when a return from subroutine was executed as shown in Fig.11.12. This led to omission ot the acceptance test following calculation of one of the difference equations. This omission was not a hazard to the system as the addition of the four parallel units is checked later in the cycle before a guidance demand is sent to the missile.

11.5. Use of Software Traps.

Some microprocessors, including the Z8000, have built in software traps to detect potentially hazardous situations, in addition to a software interrupt call for user software. The use of these traps was described in Chapter 7.5.; using this technique the system was run using the faults listed in Table 11.1.

In addition to those recovered from by the basic Recovery Block, numbers one and five did not cause a system crash using the technique of reading the process number and returning control to the appropriate alternate routine.

11.6. Addition of Watchdog Timer.

A watchdog timer, as previously described in Chapter 7.3., was added to the structure of the Recovery Block within the digital controller. The remaining faults from Table 11.1. (instructions most likely to cause a system crash) that were not recovered from using the mechanisms in Chapter 11.4. (Basic Recovery Block) and 11.5. (Software Traps) were introduced into the controller software. A time out occurred on each occasion leading to entry of the alternate routine. No degradation in system performance resulted from the injection of these faults.

11.7. Global Recovery.

Under fault conditions the 1553B bus controller may request data and repeatably receive a busy response. Alternatively the failure of a remote terminal may lead to the message error bit being set and the suppression of the status word. In a real time system, the controller cannot continually accept this situation and must take steps to maintain the integrity of the system. This section describes how the system can deal with the transient failure of a remote terminal, in this case the terminal attached to the target tracking processor. The permanent failure of this processor is covered in Chapter 11.8.

11.7.1. Transient Failure and Recovery.

Consider the transient failure of the remote terminal belonging to the target tracking processor for one system cycle. The transient failure was simulated using the 1553B protocol fault injection interface described in Chapter 10.2.6. At the required time of failure, the frame length was adjusted to twenty one bits for a single cycle only. The target tracking process is a slow moving one, therefore the last correct value received by the controller is a reasonable estimate of the true position of the target.

The recovery of the system is explained by following the run of the above failure, with the aid of Fig.11.13. On the fault cycle, the bus controller receives an invalid status word each time a request for data is made. This is allowed to occur a maximum of four times; this figure being set by the maximum latency allowed in the system. At this stage the digital controller assumes that the remote terminal is not going to reply and enters an alternate routine. This routine is a stepping stone between fault free operation and the permanent failure of a remote terminal or subsystem, thus a transient failure is first assumed.

11.7.2. Example of Recovery.

For this example the system entered the alternate routine and the last correct data from the target tracking process was used. In addition, a counter was updated for the purposes of counting the number of times the alternate routine was entered; a maximum value of five was allowed before a permanent failure was diagnosed. The use of the last correct data corresponds to the target azimuth position which is used as the input to the missile guidance loop.

On the next cycle the target tracking processor responded correctly to the bus controller's request for data, and the target azimuth was sent from the remote terminal to the controller. This cycle and the following cycles were successfully completed.

The fault was induced in a cycle on which the target azimuth did not change, and as recovery took place the missile angle was exactly as in the fault free operation. If the fault had occurred on a cycle when the target azimuth had changed, the digital controller would have used the previous value on the faulted cycle and the true value on the next cycle. This would have resulted in the step change in target azimuth appearing 1/120th second later than it should have done.

Chapter 12. Standby Processing Systems.

The previous chapter demonstrated the improvement of availability that can be obtained in a distributed processing system under fault conditions.

Consider now a system which decomposes into a given number of processor subsystems due to factors such as complexity, allowed latency, distribution of system peripherals and prevention of propagation of faults. How then is the decision made to include a further processor to increase system availability and performance under fault conditions and what function will it undertake.

The decision to add an extra microprocessor subsystem and the amount of fault tolerance within the other microprocessor subsystems, is based largely on system requirements, i.e. how is the system expected to operate under certain specified conditions. The operating conditions may include environmental conditions such as EMP radiation, permanent or transient fault conditions, and difficulty of maintenance whilst in field use.

The processing power of an additional microprocessor system may be used for task swapping and/or health monitoring; these functions are described below.

12.1. Task Swapping.

The concept of using a standby microprocessor system is not a new idea, however it is not sufficient to obtain a better performance under fault conditions. The additional processor may need to gain access to peripherals or transducers within the system, and this access will depend upon the physical system distribution and the availability of transducers. The use of the terminal attached to the standby unit as a remote terminal or as a standby bus controller will depend upon the number and nature of the remote terminals and the attached subsystems, and the requirement for continued system operation. For example, it may be imperative that a bus controller failure does not cause system failure.

12.2. Health Monitoring.

In many real time systems it is important to give an operator confidence that the system is functioning fully or in a degraded mode. The importance of this confidence may vary depending upon environment and skill level of operator. In order to gain confidence that the system is operational it is necessary to carry out routine health monitoring; this monitoring must be integrated into the design of the system. In the system described the digital controller could send its immediate outputs of the difference equations to a standby processor on a regular basis. The reception of this data can then be used for health monitoring, that is, a signal from the bus controller to confirm the functional state of the system. In the event of a bus controller or digital controller subsystem, the standby processor can use the last valid set of intermediate outputs rather than restart the difference equations from zero.

12.3. Use ot Field Test Data.

The system requirements may or may not be sufficient to determine the system configuration; additional data in the form of field test data, if available, can be used for the basis of the decision. This field test data can be gathered, if possible, from existing equipment using for example the same transducers and/or operating in similar environmental conditions. From this data, it may be deduced, for example, that transient faults predominate or that a certain transducer is critical to the operation of the system or that the communications link is prone to burst errors. The field test data can be used to decide whether the system operational requirements are likely to be met with a certain configuration and determine the level of fault tolerance within the subsystem and the need for a standby microprocessor system.

Having considered aspects of a standby processing subsystem, the following sections describe the recovery process that takes place following a subsystem failure and the associated achieved performance.

12.4. Failure of a Remote Terminal.

The addition of a fourth processor subsystem was provided in order that system recovery could take place when a complete processor subsystem failed. This section describes the recovery that takes place following a remote terminal failure whilst the system is tracking a target.

During the four processor study the raw target data was loaded into the memory of the missile processor, as this processor is assumed fault free. This involved the building of an additional memory card whose circuit diagram, layout diagram and parts list can be found in Figs.12.1. and 12.2. and Table 12.1. The placement of this raw data within the memory of the missile processor enabled the system to obtain target data even in the presence of the digital controller or target tracking processor failure. This involved a small modification to the software, that is on each cycle the bus controller has to get the raw data and give it to the target tracking processor. This involved a time overhead but it was small compared to the cycle time, thus having no effect on system performance. The arrangement of the four processor subsystems and the software is shown schematically in Fig.12.3., where the fourth processor contains a copy of the target tracking process and is idle during fault free operation.

12.4.1. System Recovery.

Consider the permanent failure of the target tracking processor and the associated recovery. For this example, the failure of the target tracking processor results in a busy reply when a request for data is made. On the first cycle of the failure, a maximum number of busy status returns are received, leading to entry of an alternate routine shown in Fig.12.4. The last correct value of target azimuth is used and the guidance demand calculated. System considerations determine that no more than six consecutive entries of the first alternate routine were allowed. The fault, being permanent, after five cycles causes the system to enter the alternate routine for a sixth time and then assumes a permanent failure.

The second alternate routine is then entered and this effects the use of the fourth processor to take over the failed processor's function. On the first cycle in this alternate routine, the digital controller has to give the standby processor sufficient information to take over the failed function. In this case, the bus controller sends the radar azimuth position and the azimuth on which the target lies. The reception of these variables by the standby processor acts as a wake up signal, with these variables being used as a starting point of the function.

It is assumed that time is limited on this sixth cycle and so the digital controller again uses the last stored value of the target azimuth. On subsequent cycles the digital controller enters the alternate routine, sends raw data to and receives target azimuth positions from the standby processor.

12.4.2. System Performance.

If the failure of the target tracking processor occurs at least six cycles before a change in target azimuth then no difference in the resultant missile angle is obtained. The digital controller has no knowledge of the targets range or velocity characteristics and so a period of graceful degradation occurs for a period less than one second until the standby processor identifies the target.

If the failure of the target tracking processor occurs less than six cycles before the target azimuth position is due to be updated, then the resulting missile angle plot will be different from that of the unfaulted one. This is due to the effect which can be seen schematically in Fig.12.5. The standby processor does not identify a target on a particular cycle until approximately one second after the fault, and uses the target azimuth value prior to the fault.

Two runs were carried out with a failure of the target tracking processor occurring less than six cycles before the target azimuth was due to change. In the first run, the fault was introduced at one second after the start of the run. The resultant missile angle plot can be found in Fig.12.6. and shows that only the gathering phase is affected and the missile still enters the terminal phase successfully. A maximum deviation of 2° was recorded with a performance index of 46.5(seconds. degrees)^2. The second run involved a fault at approximately eight seconds, i.e. during the terminal phase. The resultant missile angle plot can be found in Fig.12.7. and shows that the system regained tracking within three seconds, giving a performance index of 154.9(seconds. degrees)^2, and a maximum deviation ot 1.44°. As the angle was within three degrees of the true unfaulted angle at eleven seconds, then the run was considered to be successful. Eight seconds from the start of the run, was found to be the latest time that such a fault could occur without affecting mission success.

12.5. Failure of a Bus Controller.

The failure of the target tracking processor during system operation did not cause system failure due to recovery taking place with the aid of a standby processor. Intuitively, the failure of the bus controller is likely to have a much greater effect on system performance. This section shows by way of examples how the recovery from such a failure can take place and its effect upon system performance.

The configuration of the four processor systems, was as shown in Fig.12.8. with the standby processor idle under no fault conditions. Consider then the failure of the digital controller whilst tracking a target.

The function of the digital controller is to execute a number of difference equations to calculate the guidance demand of the missile. If another processor has to take over then it is advantageous to use a good estimate of the past values of the four parallel units rather than restart the difference equations from zero. The outputs of the four parallel units can be seen in Fig.11.10. which shows that the best estimate for previous outputs is in fact zero.

It was assumed that the failure of a bus controller would result in a prolonged period of inactivity or a prolonged period in which invalid commands are being transmitted on the bus. This period was detected by

the failure to retrigger a monostable by the valid command sync pulse derived from the bus monitor. The output of the monostable was then polled by the microprocessor subsystem to detect the bus controller failure.
For the purposes of the study the minimum period of inactivity was set to four milliseconds from the receipt of the last valid command sync pulse, this being shown in Fig.12.9.

Thus the detection mechanism consisted of a retriggerable monostable which was continually retriggered during normal bus operation giving a logical '1' output. Following bus controller failure the monostable is not triggered and the output falls to a logical '0'.

12.5.1. Use of Bus Monitor.

The failure of the bus controller was carried out by the use of the non-maskable interrupt mechanism as previously described. The subsystem processor (digital controller) was put into a halt condition, thus taking no further part from the time of failure to the end of the run. In practice the failed bus controller must not be allowed to issue further commands, after it has deemed to have failed by a bus monitor. This can be carried out, as shown schematically in Fig.12.10. by the use of a discrete which disables the output of the bus controller. This discrete is set by the bus monitor on detection of a prolonged inactive bus period.

Having detected prolonged bus inactivity the bus monitor then assumes bus control. The standby processor must then obtain the target azimuth from the target tracking processor and read the missile angle. The missile to target error angle is used as input to the difference equations, setting previous inputs equal to the present input, and the previous outputs of the four parallel units equal to zero. The system then continues as normal during which time coverage is still given by the target tracking processor.

12.5.2. Effect of Failure on Performance.

The effect of the bus controller failure on the missile angle depends upon when the failure occurs during the run. The greatest deviation in in the missile angle occurred when the failure took place in the gathering phase. This occurred due to the starting up of the difference equations immediately after bus controller failure. During the gathering phase the target to missile error angle is large and not equal to zero, even if zero is the best estimate. A failure at one second after the start of the run results in the missile angle plot as shown in Fig.12.11. This shows a large deviation from the true missile angle (9.5°) with recovery taking about eight seconds, resulting in a performance index of 738.5(seconds. degrees)^2. This large deviation affects the missile angle during the gathering phase but shows that tracking still occurs before the target is reached.

The effect of the failure occurring later in the gathering phase results in a smaller excursion from the true missile angle as can be seen from Figs.12.12. and 12.13., which show the effect of a failure at two seconds and four seconds respectively. The failure at two seconds gives a maximum deviation of 4.01° with a performance index of 140.5(seconds. degrees)^2, whilst the failure at four seconds resulted in a maximum deviation of 2.65° and a corresponding performance index of 128.9(seconds. degrees)^2.

During the terminal phase of the missile, the missile to target error angle is small, and the outputs of the four parallel units are close to zero. Thus if a failure occurs during this phase the effect of setting the parallel outputs to zero (in the standby processor) is likely to be less than that in the gathering phase. This is likely to result in a shorter recovery time and a smaller excursion from the true missile angle. Failure of the bus controller was carried out at seven, eight and nine seconds after the start of the run, giving maximum deviations of 2.35°, 1.99° and 1.24° respectively. The resulting plots can be found in Figs.12.14., 12.15. and 12.16., these represent performance indices of 39.1, 20.5 and 12.7(seconds, degrees)^2. The graphs show that the time to recovery and the maximum excursion are less than that in the gathering phase and that mission success is not affected by a bus controller failure even in the terminal phase of the missile.

Chapter 13. Distributed Processing Conclusions.

The implementation of fault tolerant techniques within a distributed processing environment has resulted in an increase in availability under extreme operating conditions. However, it must be stressed that redundancy does not automatically increase the reliability ot a system. A poor implementation of a fault tolerant technique may actually result in a decrease of system reliability.

13.1. Review of Design Philosophy.

The use of a Recovery Block within subsystems which form part of a distributed system provides recovery on a local basis. This ability to recover locally has led the author to establish a design philosophy for message passing between processors. This philosophy is based on testing data at the point of maximum information, i.e. at the point of transmission of the message, and the absence of testing data on reception. The testing of data is carried out by an acceptance test prior to transmission; the data is assumed to be valid if it is received correctly with respect to the particular communications protocol.

The absence of reception of expected data leads to another principle, i.e. that message transfers proceed only in one direction. If a message fails to arrive then the receiving subsystem must not attempt to diagnose the failure to transmit; instead it must initiate global recovery after a predetermined time period. If the receiving device were allowed to attempt fault diagnosis of the transmitting subsystem a loop would be closed around the communications link, and the system would become more complex and probably more unreliable.

13.2. Distributed Processing Recovery.

The distributed processing research has shown that by using the Recovery Block as a basis, transient and permanent faults can be recovered from generally without a severe loss of performance. System recovery was shown to take place whilst real time control was being performed, without massive redundancy as in triple modular redundancy.

The faults injected were divided into two groups, i.e. data corruption type faults and system crash type faults. The distributed processing system without a recovery mechanism was still able to track targets when the data corruption type faults were injected during the gathering phase. The effect of data corruption in the terminal phase let to a high probability of missing the target being tracked.
By definition this sytem was unable to recover from system crash type faults.

13.2.1. Local Recovery.

The implementation of the basic Recovery Block within the distributed processing system ensured that recovery took place when data corruption type faults were injected into the target tracking and digital controller processes. This implementation was unable to recover from system crash type faults; this confirmed the results of the single processor study. The use of the time domain in the form of a watchdog timer and the use ot system traps for illegal conditions led to recovery from the system crash type faults.

13.2.2. Global Recovery.

If local recovery from a particular fault was not possible, then global recovery was shown to maintain the system functional. Global recovery was performed by the use of an alternate routine in the master processor subsystem, and is necessary if transient faults prevent the master from receiving valid data. The use of local recovery means that there is a high probability that the processor's communication interface is loaded with data, but cannot guarantee correct communication of data.
Under these conditions, global recovery is necessary to ensure valid data and continued system operation.

13.2.3. Use of a Standby Processor.

If system availability is required to be high then the use of a standby processor system may be justified. The failure of a slave subsystem was performed and dynamic task swapping was shown to give good results when the system was tracking a target. The task swapping was initiated when a counter exceeded a predetermined limit within an alternate routine in the master processor subsystem. This was followed by enabling the standby processor with the necessary starting values. The failed subsystem took no further part and all communication with the particular function was made to the standby processor. This type of failure did not affect system success provided it occurred more than three seconds from the target.

In a master/slave system, the master is critical for continued operation and high availability. A bus controller failure was carried out which did not lead to a system crash due to bus inactivity detection circuitry within the bus monitor. Assuming that the bus controller fails quietly, i.e. no bus communication traffic, then this effect can be used to initiate take over of bus control. The new bus controller must ensure that the failed bus controller takes no further part in the operation of the system. The results showed that failure of the bus controller, even in the terminal phase of the missile did not affect the objective to hit a target. System performance was only slightly impaired as shown by the low performance indices recorded in the terminal phase, as shown in Chapter 12.5.

The take over of control by the bus monitor was fast and occurred within one system cycle. The degradation in performance was due to the settling of the digital controller's difference equations in the new master subsystem. This performance can be improved if the intermediate outputs of the difference equations are regularly transmitted to the bus monitor. The transmission of these outputs can also act as a health monitoring signal to the bus monitor. In the event of the bus controller not failing quiet, the absence of a health monitoring signal can be used to signify a failure of the bus controller, without waiting for a quiet period on the bus.

13.3. Further Work.

The modelling of hardware reliability is well established, unlike the field of software reliability modelling which is a comparatively new one. However, in the view of the author the problem is being tackled incorrectly since the all important point is the reliability of the system. Few researchers (if any) have tackled the self imposed problem of combining hardware and software models to give a system reliability model. This area needs consideration before too much time is spent on developing software reliability models.

The system described in this thesis was operated without need for an operating system. Some real time systems may require a kernel to supervise the operation of parallel co-operating processes. Such a kernel would also require fault tolerance for high reliability, Further work is required to establish the implementation of a Recovery Block structure within such a system. It is likely that the kernel would be considered as the highest level of software and perform acceptance on processes either running or to be run.

The single processor study involved applying mainly single faults with a small percentage of double faults. This was considered to be sufficient within the time available, however further work could be usefully spent by studying the implementation of a Recovery Block structure under multiple fault conditions. An important area for investigation is the development of robust software specifically for areas where input data is likely to be corrupt.

In the distributed processing study, a standby processor was effectively used for continued systems operation under the conditions of a failed subsystem. Under normal operating conditions the standby processor is idle and could be used for system health monitoring, that is to monitor and record the state of the system.

Chapter 14. Towards an Integrated Approach to Design.

The approach used in this report was to investigate different strategies including the assessment of their performance in order to arrive at a system with high availability under prescribed fault conditions. The experience gained from the study is used here to discuss guidelines for the design of a reliable system. In addition, these guidelines have been applied to the design of a single microprocessor target tracking system; this design is illustrated using a MASCOT methodology.

14.1. Guidelines for Design.

The use of redundancy is often necessary in order to achieve system reliability and availability requirements. However redundancy must be applied methodically to ensure that system complexity is not unnecessarily increased. This section presents guidelines for the design of reliable systems.

14.1.1. Functional Decomposition.

The functional decomposition of a system is an essential feature of the system recovery strategy. The factors to be considered are:

1. Separation by function or process.
2. Interprocess communication kept to a minimum.
3. Consideration of physical locality of functions.
4. Functions need to be a manageable size for a complete understanding of the total system.

14.1.2. Recovery Block.

The use of a Recovery Block structure must be justified within the system to be designed. Consideration should be given to the overhead incurred with relation to the increase in availability obtained. The single processor study gave an increase from 5% to 42% availability (with fail safe). This must be weighed against the overhead in sottware resulting from the use ot the structure; a figure of 30% additional software was found to be typical.

14.1.3. Watchdog Timer.

The use of the time domain for implicit fault detection was considered to be an essential feature of any real time system. The watchdog timer is simple in hardware terms, consisting of a programmable timer which can set an interrupt flip flop. An interrupt service routine must be written to determine the process which was being performed at the time of the fault and transfer control to the relevant re-entry point. Results from the single processor study showed an availability of 85%, an increase of 43% over the basic Recovery Block structure.

14.1.4. Run Time Overhead.

The overhead in time, incurred by using a Recovery Block structure is dependant upon the complexity of the acceptance tests and the environment in which the system operates. If the environment is noisy electrically then transfer of control into alternate routines is likely to be common.

14.1.5. System Traps.

Any unused software or hardware traps available within the processor must be restored to the same address as that for the hardware timer. A log of fault interrupt causes can be kept for continuous monitoring and maintenance purposes.

14.1.6. Reversionary Modes.

Systems design must take account of reversionary modes of operation upon fault detection. A safe shutdown of the system is often desirable if a hazardous condition is detected.

14.1.7. MASCOT ACTIVITY CHANNEL POOL (ACP) Diagram.

An initial design is illustrated using an ACP diagram, which shows the Activities of the system and the Intercommunication Data Areas. The reader is referred to Ref.6. for information on MASCOT. An inadequate decomposition of the system will result in a large ACP diagram with highly interconnected activities.

The overall system design is illustrated as a hierarchical set of ACP diagrams. Decomposition is carried out to a depth necessary to achieve a reasonable level of functional modularity.

14.1.8. Fault Scenarios.

Having decided upon a hierarchical set of ACP diagrams then system designers should study the diagrams to identify situations which might compromise safe system operation. If a hazardous situation is identified then a fail safe mechanism or alternative strategy is necessary.

14.1.9. Design Reviews.

Design Reviews should be carried out to ensure that the system specification requirements are adequately stated and can be feasably met. A design Review should cover the following points:

(i) Clarity of software structure.

(ii) Tolerance of software to hardware errors.

(iii) Design proving requirements.

(iv) Requirements for configuration control.

(v) Safety.

(vi) System development tools.

(vii) Acceptance procedures.

(viii) Reversionary modes of operation.

(ix) Software/Hardware trade offs.

14.1.10. Structured Walkthroughs.

The structured Walkthrough is similar to a Design Review except that it is carried out with greater frequency. It is concerned with the design of a subsystem or part of a subsystem and covers the following points:

(i) Function.

(ii) Clarity of structure.

(iii) Speed of operation.

(iv) Test requirements.

(v) Fault detection and recovery.

(vi) Size of software.

14.1.11. Testing.

The use of a Recovery Block structure has the advantage that testing of software can be modular and more thorough thus removing a greater percentage of design errors. In top down testing, the top level is tested first, a lower segment is added and the combination tested. This is repeated down to the lowest level. Dummy segments temporarily replace the segment subordinate to the segment under test. These dummy segments can vary in complexity and may return constants or may be a primitive version of the segment being simulated. To enhance structured programming the length of a segment should be limited to a mangeable level, say fifty statements to enhance readability and comprehension whilst minimising page turning. Usually each segment will correspond to one function and can be implemented as a procedure with a descriptive name corresponding to the function. Thus the use of small segments makes programs easier to extend and maintain; reliability is further enhanced since test plans for the segments are easier to specify and execute.

14.2. Single Processor System.

Having discussed guidelines for reliable systems design, this section describes the initial design of a single microprocessor target tracking system. It is assumed here that the microprocessor to be used is capable of the real time processing necessary.

14.2.1. Functional Decomposition.

Using the factors detailed in Section 14.1.1. it was decided to use the same decomposition as previously used. However the sub tasks will no longer be processed in a sequential order, due to the operation of the system in a MASCOT environment.

14.2.2. Recovery Block.

It was considered that the use of a Recovery Block structure could be justified in order to obtain a high availability. The inclusion of a Recovery Block structure is not sufficient to increase system availability; it is necessary to ensure that the implementation is robust. The implementation of the Recovery Block structure on a particular processor system will result in a particular overhead, which is application dependent. The estimated overhead in software and hardware can be weighed against the increase in availability obtained. At present, as far as is known, this study represents the only source of information on the increase in availability that can (not necessarily will) be obtained by using a Recovery Block structure.

14.2.3. Watchdog Timer.

The introduction of a watchdog timer can be justified here, as it involves little overhead in software and hardware terms.

14.2.4. Run Time Overhead.

The target tracking system is operated with an angular separation of 12^o. The time taken for the processing will depend upon the processor chosen. A correct choice of processor will allow a Recovery Block structure to be used.

14.2.5. Trap Areas.

The use of trap areas between code segments does not necessarily result in an increase of availability. However, this feature can be effectively used for safety purposes, that is to ensure that a routine is correctly entered. It is considered sufficient for this system to include a trap area immediately before each primary routine.

14.2.6. Reversionary Modes.

The reversionary modes of operation in the target tracking system simply consist of alternate routines relevant to the particular process. The system is shutdown if any alternate routine is entered on four consecutive cycles. This is considered to be the point at which the system can no longer give valid outputs. No hazardous states exist within the target tracking system.

14.2.7. MASCOT ACP Diagram.

The top level ACP diagram for the target tracking process is shown in Fig.14.1. Whilst the system is in a standby state, i.e. SEARCH mode, then time is available for checking of system hardware. Using a priority scheduler then the activity for hardware checking can run at the lowest priority. The design of an activity scheduler is not considered here.

Chapter 15. Overall Review of Achievements.

This chapter reviews the research study in terms of the objectives set out in Chapter 1.1. The study has conclusively shown that the availability of a system can be improved by a combination of measures as outlined in the following paragraphs.

For completeness the constituent parts of the main objective are repeated below, together with reference to the relevant chapters where they are achieved.

(a) 'To establish good design practices based upon a practical rather than a mathematical approach'.

Guidelines to design are discussed in Chapter 14 which presents an integrated approach. This approach is applied to the design of a target tracking system as described in Chapter 14.2.

(b) 'To establish a simple but obvious structure for system recovery'.

The Recovery Block was shown to be a basis for the design of reliable real time systems as described in Chapters 7 and 8.

(c) 'To establish design criteria for reliable inter-task communication within a single processor'.

The integrity of data was improved by a method wherèby system variables were not updated until the appropriate acceptance test had been successful. The system variables were then passed to the next task by the use of CPU internal registers as described in Chapter 7.

(d) 'To establish a design philosophy for message passing between microprocessors in a distributed system in order to inhibit the propagation of faults'.

> The concept of checking data before passing it to the next task was extended to the distributed processing environment where the receiving processor accepts data as valid unless otherwise indicated by the transmitting processor. This philosophy is described in Chapter 9 with results in Chapter 11.

The overall conclusion of the research study was that for reliable systems operation, fault recovery must be localised to minimise the propagation of faults to the next task in a single processor system or to another processor in a distributed system. The conclusions for the single processor study are presented in Chapter 8, whilst the distributed processing conclusions are presented in Chapter 13.

The initial objectives were to investigate recovery from transient faults; however opportunity was taken to extend the study to investigate failures of a catastrophic nature whereby a subsystem fails permanently. As described in Chapter 12, the strategy adopted in this respect was to introduce a standby processor in a task swapping mode. Conclusions drawn from the results obtained are presented in Chapter 13.

Acknowledgements.

The author would like to thank British Aerospace and the Science and Engineering Research Council for support and funding of the research study. The author would also like to thank Prof. J.F. Eastham for the use of the laboratory and Dr. B.A. White for his guidance and assistance throughout the study. Thanks are also due to Mr.M.G. Brown and Mr. S.C. Dunn of British Aerospace for their assistance with the research,and Mr. R.V.S. Penfold and his team, also from British Aerospace, for hardware support.

Thanks are finally due to Mrs. P. Jackson for the typing of this thesis.

References

1. FISCHER K.F., WALKER M.G.: 'Improved Software Reliability Through Requirements Verification'; IEEE Trans. Reliab. R-28, pp. 233 - 240, August 1979.

2. MELLIAR-SMITH P.M., RANDELL B.: 'Software Reliability: The Role of Programmed Exception Handling', SIGPLAN Notices 12(3). pp.95 - 100, March 1977.

3. BALZER R., GOLDMAN N., WILE D.: 'Informality in Program Specification', IEEE Trans. Software Engr. SE-4, pp. 94 -103, March 1978.

4. MYERS G.J.: 'Software Reliability', John Wiley, 1970.

5. ROSS D.T., 'Structured Analysis (SA): A Language for Communicating Ideas', IEEE Trans. Software Engr. SE-3, pp. 16-34, January 1977.

6. The Official Handbook of MASCOT, MASCOT Suppliers Association, December 1980.

7. MYERS G.J.: 'Reliable Software through Composite Design' Van Nostrand Reinhold Company, 1975.

8. PARNAS D.L.: 'On the Criteria to be Used in Decomposing Systems into Modules', Communications of the ACM 15(2), pp. 1053 - 1058, December 1972.

9. FUSSELL J.B., POWERS G.J., BENNETTS R.G.: 'Fault Trees - A State of the Art Discussion', IEEE Trans. Reliab. R-23, pp. 51 - 55, April 1974.

10. LONDON R.L.: 'Proving Programs Correct: Some Techniques and Examples', BIT 10, 1970.

11. WENSLEY J.H. et al: 'SIFT: Design and Analysis of a Fault Tolerant Computer for Aircraft Control', IEEE Proceedings Vol.66, pp.1240-1255, October 1978.

12. PETERSON W.W., WELDON E.J.: 'Error Correcting Codes' 2nd Ed., MIT Press 1972.

13. CHEN L., AVIZIENIS A.: 'N-Version Programming: A Fault Tolerance Approach to Reliability of Software Operation', Dig. FTCS-8, Eighth Ann. Intl. Conf. on Fault Tolerant Computing, pp. 3-9, 1978.

14. RANDELL B.: 'System Structure for Software Fault Tolerance', IEEE Trans. Software Engr. SE-1, pp. 220-232, June 1975.

15. JELINSKI Z., MORANDA P.: 'Software Reliability Research, Academic Press, pp. 465-468, 1972.

16. SCHOOMAN M.L.: 'Probabilistic Modes for Software Reliability Prediction', Int. Symp. Fault Tolerant Computing, pp.211-215, June 1972.

17. MUSA J.D.: 'A Theory of Software Reliability and its Application', IEEE Trans. Software Engr. SE-1, pp.312-327, September 1975.

18. SCHICK G.J., WOLVERTON R.W.: 'Assessment of Software Reliability', Proc. Operations Res. pp.395-422, 1973.

19. LITTLEWOOD B., VERRALL J.L.: 'A Bayesian Reliability Model with a Stockastically Monotone Failure Rate', IEEE Trans. Reliab. R-23, pp.1o8-114, June 1974.

2o. QUINN M.D., RICHTER D.: 1980 IEEE Test Conference, pp. 238-253, 1980.

21. MOORE W.R.: Electronic Letters Vol.15 No.22, pp.722-724, 1979.

22. GARNELL P., EAST D.J.: 'Guided Weapon Control Systems' Pergamon Press, 1977.

23. KATZ P.: 'Digital Control Using Microprocessors', Prentice/Hall, 1981.

24. Am96/4016 Users Manual, Advanced Micro Devices, 1979.

25. TMS 9900 Family System Development Manual, Texas Instruments, Bulletin MP702, p.58, 1977.

26. BERGLAND G.D.: 'A Guided Tour of Program Design Methodologies', Computer, pp.19-37, October 1981.

27. HARTLEY M.G. ed.: 'Digital Simulation Methods', Peter Pereguins Ltd., 1975.

28. MIL-STD 1553B: Aircraft Internal Time Division Command/ Response Multiplex Data Bus, Department of Defence, September 1978.

29. HD-15530 CMOS Manchester Encoder - Decoder, Harris Corporation, 1978.

30. HD-15531 CMOS Manchester Encoder - Decoder, Harris Corporation, 1978.

31. Z8001/Z8002 Product Specification, Zilog., 1979.

32. Principles of Operation AmZ8001/2 Processor Instruction Set, Advanced Micro Devices, Am-PUB086, 1979.

33. Principles of Operation AmZ8001/2 Processor Interface, Advanced Micro Devices, AmPUB089, 1979.

APPENDIX A

Digitisation of Guidance Loop.

In the following derivations the sampling period of the digitised system is 1/30 second.

A.1. Digital Controller: $G_1(z)$

$$G_1(z) = (1 - z^{-1}).Z\left(\frac{1}{s}.\ \frac{10(s+1)(s+1)(s+0.5)}{s(s+3.16)(s+3.16)}\right)$$

$$= (1 - z^{-1}).Z\left(\frac{1}{s}.\ G_1(s)\right) \qquad \text{..... (A.1.)}$$

By Partial Fractions

$$\frac{G_1(s)}{s} = \frac{1.685972102}{s} + \frac{0.500721038}{s^2}$$

$$+ \frac{8.314027898}{(s+3.16)} - \frac{12.42839289}{(s+3.16)^2} \qquad \text{..... (A.2.)}$$

Then $Z.\left(\frac{G_1(s)}{s}\right) = \frac{1.685972102z}{z-1} + \frac{0.016690701z}{(z-1)^2}$

$$+ \frac{8.314027898z}{z - 0.900024464} - \frac{0.372861921z}{(z - 0.900024464)^2} \qquad \text{... (A.3.)}$$

Finally $G_1(z) = (1 - z^{-1}).Z\left(\frac{G_1(s)}{s}\right)$

$$= 1.685972102 + \frac{0.016690701}{(z-1)}$$

$$+ \frac{8.314027898(z-1)}{z - 0.900024464} - \frac{0.372861921\,(z-1)}{(z - 0.900024464)^2}$$

$$= \frac{10 - 29.18785963z^{-1} + 28.39590856z^{-2} - 9.207881866z^{-3}}{1 - 2.800048928z^{-1} + 2.610092963z^{-2} - 0.810044035z^{-3}} \qquad \text{..... (A.4.)}$$

A.2. Missile Autopilot: $G_2(z)$

$$G_2(z) = (1 - z^{-1}).Z \quad \frac{1}{s}.\left(\frac{144}{s^2(s^2 + 14s + 144)}\right)$$

$$= (1 - z^{-1}).Z \left(\frac{1}{s}.\ G_2(S)\right) \qquad ..... (A.5.)$$

By Partial Fractions

$$\frac{G_2(s)}{s} = \underline{0.002507716} - \frac{0.097222222}{s^2} + \frac{1}{s^3}$$

$$- \frac{(0.002507716s - 0.062114198)}{s^2 + 14s + 144} \qquad ..... (A.6.)$$

Then $$Z.\left(\frac{G_2(s)}{s}\right) = \frac{0.002507716z}{z - 1} - \frac{0.00324074z}{(z - 1)^2}$$

$$+ \frac{0.000555555z\ (z + 1)}{(z - 1)^3}$$

$$- \frac{(0.002507716z^2 - 0.003037504z)}{z^2 - 1.500869446z + 0.627089085} \qquad ..... (A.7.)$$

Finally $$G_2(z) = (1 - z^{-1}).Z \left(\frac{G_2(s)}{s}\right)$$

$$= 0.002507716 - \frac{0.00324078}{z - 1} + \frac{0.000555555\ (z + 1)}{(z - 1)^2}$$

$$+ \frac{(z - 1)(- 0.002507716z + 0.003037504)}{z^2 - 1.500869446z + 0.627089085}$$

$$= \frac{- 0.000903767z^{-1} + 0.002798632z^{-2} - 0.002670325z^{-3} + 0.000915696z^{-4}}{1 - 3.500869446z^{-1} + 4.628827977z^{-2} - 2.755048263z^{-3} + 0.627089085z^{-4}} \qquad ..... (A.8.)$$

APPENDIX B

The Z8000 Microprocessor.

This appendix contains a brief description of the Z8000 microprocessor; further information can be obtained from Refs. 31, 32 and 33.

B.1. Architecture.

The Z8000 is a single chip 16 bit microprocessor using N-Channel MOS technology and provides a multiplexed data/address bus. The Z8000 CPU is at present offered in two versions: the Z8001 segmented version and the Z8002 non-segmented version; future versions will include a virtual memory capability. The Z8001 can directly address 8 megabytes of memory, whereas the Z8002 directly addresses 64 kilobytes. The two operating modes of the microprocessor, system and normal modes, and the distinctionbetween code, data and stack spaces within each mode allows memory extension up to 48 megabytes for the Z8001 and 384 kilobytes for the Z8002.

The Z8000 CPU contains sixteen 16 bit general purpose registers, a status register (Flag and Control Word), a program counter, a program status area pointer and a refresh counter register.

B.2. Interrupts and Trap Structure.

The Z8000 provides three types of interrupts (non maskable, vectored and non vectored) and four traps (system call, unimplemented instruction, privileged instruction and segmentation trap). The segmentation trap is only available on the Z8001.

When an interrupt or trap occurs, the current program status is automatically pushed onto the system stack. The program status consists of program counter, the Flag and Control Word, and a 16 bit identifier. The identifier contains the reason or source of the trap or interrupt. After saving the current program status, the new program status is automatically loaded from the program status area in memory which is directed to by the program status area pointer.

B.3. Memory.

The Z8000 uses four control signals in association with four status signals during memory read or write cycles. The multiplexed bus contains a valid address on the rising edge of Address Strobe ($\overline{AS}$). The Data Strobe signal ($\overline{DS}$) is used to indicate either valid data on a write cycle or that the CPU expects valid data on a read cycle. A memory request signal ($\overline{MREQ}$) is active during all memory cycles.

Consider first a memory read cycle, the timing diagram is shown in Fig.B.1. which assumes that the memory used has an access time comparable to one clock period. Slower memories can be used by the addition of wait states. At the beginning of the cycle the Read/$\overline{Write}$ (R/$\overline{W}$) signal goes high. The rising edge of $\overline{AS}$ indicates a valid read address, data can be placed on the bus after $\overline{DS}$ becomes active and is read by the CPU on the rising edge of $\overline{DS}$.

On a memory write cycle (shown in Fig.B.2.) the R/$\overline{W}$ line is low, and valid memory address is indicated as for the read cycle. Valid data may be taken off the bus whilst Data Strobe is low.

B.4. Input/Output.

Input/Output is carried out in a similar manner to memory accesses with the exceptions that the memory request line is not active, an automatic wait state is inserted, and the status lines indicate an I/O reference. I/O devices are addressed with a 16 bit port address.

Direct memory access (DMA) can be carried out over the Z8000 multiplexed bus during which time the bus is driven by a DMA device.

B.5. Instruction Set.

The Z8000 provides the following types of instructions:

Load and exchange
Arithmetic
Logical
Program control
Bit manipulation

Rotate and shift

Block transfer and string manipulation

Input/output

CPU control

APPENDIX C

The Micromaster.

The Micromaster was developed by the Control Group at the University of Bath, School of Electrical Engineering, for use as a microcomputer teaching aid. Its use as a teaching aid is not described here since the Micromaster was simply used as an intelligent terminal for the duration of this study.

The Micromaster contains a Z80 microprocessor and this was used to communicate with the PDP 11 through an RS232 port and with the Z8000 microprocessors through the other RS232 port. Temporary storage of system results was carried out using 32K bytes of dynamic RAM within the Micromaster.

The communication software written for the Micromaster basically consists of polling the serial interface parts but is not described within this report.

APPENDIX D

Z8002 Microprocessor Program Assembler.

This appendix describes the two pass assembler which produces an object code file and an assembly listing for the Zilog Z8002. This program runs on the PDP 11 under theRSX-11M operating system.

D.1. Statement Format.

A Z8002 assembly language statement is defined as follows:

label: opcode operand(s) comments

The label and comment fields are optional, and no continuation lines are allowed.

D.1.1. Label Field.

The label field may contain a user-defined symbol containing up to six characters, the first of which must be alphabetic. The assembler allocates the current location to the label, so that a user may make further references to the label without knowing its address. A symbol used in a label field may not be redefined in the label field of another statement.

D.1.2. Opcode Field.

The opcode field follows the label field and contains one of the following:

1. Mnemonic operation code of a machine instruction
2. Assembler directive operation code.

The opcode field is terminated by a space, tab, semi-colon when there are no operands or a carriage return when there are no operands or comments.

D.1.3. Operand Field.

The operand may contain up to four expressions or terms, depending upon the type or requirements of the opcode. The operand field must follow an opcode and can be terminated by a semi-colon when a comment is to follow or by a carriage return when there are no comments.

D.1.4. Comment Field.

The comment field is used purely to help the user or future users on the workings of the assembly language program. It may be preceded by any or more of the fields previously mentioned. The comment field has no effect on the assembly and must be preceded by a semi-colon and terminated by a carriage return.

D.2. Z8002 Expressions.

This section describes the components of legal Z8002 expressions which include the instruction set, numbers and characters.

D.2.1. Character Set

The following characters are valid in Z8002 source programs:

1. The letters A to Z.
2. The digits 0 to 9.
3. The special characters as below:

Character	Designation
(	left parenthesis
)	right parenthesis
,	comma
<SP>	space
<HT>	horizontal tab
^	up arrow
$	dollar
'	apostrophe
*	asterisk
+	plus sign
-	minus sign
.	full stop
/	slash
<LF>	line feed
<VT>	vertical tab
<FF>	form feed
<CR>	carriage return
#	hash

If any character other than those above is encountered the line being assembled will be terminated and an 'I' will occur on that line in the listing.

D.2.2. Numbers.

Numbers used in the assembly language problem may be decimal, hexadecimal, octal or binary. Any number must be preceded by '#' and one of the following: ∧O (denotes octal number), ∧H (hexadecimal), $ (hexadecimal), ∧D (decimal) or no characters.

If '#' is followed by a number then the number defaults to decimal.

Octal numbers consist of the digits '0' to '7' only.

Hexadecimal numbers consist of the digits '0' to '9' and the letters 'A' to 'F'.

Decimal numbers consist of the digits '0' to '9'.

Binary numbers consist of '0' and '1' only.

A truncation error ('T' on the assembly listing) will occur if the converted number is too large to fit into eight bits for byte operations, sixteen bits for word operations or thirty two bits for long word operations. All numbers are considered to be in two's complement arithmetic.

The binary representation of a number is not implemented for thirty two bit operations.

D.3. Assembler Directives.

These are statements which are used at assembly time for ease of programming such as set a label equal to a constant, and are non executable as far as the Z8002 microprocessor is concerned.

D.3.1. Title.

The title directive is used to print a heading on the output listing. The heading will be printed on the first line of each page of the listing.

For example,

```
TITLE      PROGRAM TO CALCULATE SQUARE ROOTS
```

The 'TITLE' directive appears in the opcode field, if omitted the title defaults to 'MAIN'.

D.3.2. Page Ejection.

Apart from the automatic page eject after 61 line counts, a form feed may also be used to cause a page eject.

D.2.3. ORG.

The location at which the machine code is to be placed may be changed by the ORG directive.

For example,

```
        label:      ORG   $3000          ; comment
```

will place the following code in memory locations starting at 3000_{16}.

D.3.4. EQU.

The EQU directive assigns a value to a symbol name, which will be used when that symbol is further encountered in the program. The directive is of the form:

```
        name        EQU        value        ; comment
```

The symbol name must appear in the label field without a following colon and cannot be re-defined within an EQU directive.

D.3.5. SET.

This is identical to the EQU directive, except that the symbol name may be redefined.

D.3.6. END.

The END directive indicates the end of the source program. It may have an optional label and/or comment field. Any statement following this directive will be ignored by Z8002.

D.3.7. DEFINE WORD.

The Define Word (DW) directive is used to set a memory location to a user determined value and is of the following form:

```
        label:      DW         value        ; comment
```

D.4. Instruction Set.

The instruction set for use with the assembler may be found in AmZ8001/2 Processor Instruction Set Manual [32], and is fully implemented for use with the Z8002 microprocessor.

D.5. Addressing Modes.

This section describes the addressing modes available for use with Z8002, see Ref.32. for details of which addressing modes can be used with each instruction.

	Addressing Mode	Example
1.	Register	R6
2.	Indirect Register	(R3)
3.	Direct Address	FRED
4.	Immediate	#4
5.	Indexed	FRED (R1)
6.	Base Address	R6 (#5)
7.	Base Indexed	R5 (R4)
8.	Program Relative	BILL

D.6. Permanent Symbol Table.

The assembler contains a permanent symbol table whose entries may be not redefined. The explanation of these symbols follows:

Symbol	Meaning
RL0	Byte Registers
RL1	
RL2	
RL3	
RL4	
RL5	
RL6	
RL7	
RH0	
RH1	
RH2	
RH3	
RH4	
RH5	
RH6	
RH7	

Symbol	Meaning
R0	Word Registers
R1	
R2	
R3	
R4	
R5	
R6	
R7	
R8	
R9	
R10	
R11	
R12	
R13	
R14	
R15	
RR0	32 bit Registers
RR2	
RR4	
RR6	
RR8	
RR10	
RR12	
RR14	
RQ0	64 bit Registers
RQ4	
RQ8	
RQ12	

Symbol	Meaning	
NZ	Not zero	Condition Codes
ZR	Zero	
NC	No carry	
CY	Carry	
PO	Parity odd	
PE	Parity even	
PL	Plus	
MI	Minus	
NE	Not equal	
EQ	Equal	
NOV	Overflow is reset	
OV	Overflow is set	
GE	Greater than or equal	
LT	Less than	
GT	Greater than	
LE	Less than or equal	
LGE	Logical greater than or equal	
LLT	Logical less than	
LGT	Logical greater than	
LLE	Logical less than or equal	
Blank	Unconditional	
C	Carry	Used in Flag instructions such as SETFLG
Z	Zero	
S	Sign	
PV	Parity/Overflow	
V	Vectored interrupt	Enable/disable interrupts
N	Non vectored interrupt	

Symbol	Meaning	
FCW	Flag and control word	Used in LDCTL instruction
REF	Refresh register	
OFF	NPSAP offset	
SP	Stack Pointer	
FLGB	Flag byte - used in LDCTLB instruction.	

D.7. Using Z8002.

The assembler may be run as follows:

```
> Run Z8002.
Z8002 > FILE, FILE = FILE
```

Where FILE = program to be assembled which must have a

The above command generates an object file and a list file which is sent to the printer. Only an object file is created if the command line is as follows:

```
Z8002 > FILE = FILE
```

D.8. Error Codes.

Two types of error can occur.

1. Errors which halt assembly are as follows:

? BAD SWITCH ?	The switch specified was not recognised.
? TOO MANY INPUT FILES ?	Only one input file may be processed at a time.
? NO INPUT FILE ?	No input file was specified.
? TOO MANY OUTPUT FILES ?	Too many output files were specified.
? WRITE ERROR ?	An error occurred when attempting to write to output file.
? SYMBOL TABLE FULL ?	All the available symbol table space has been used.
? INTERNAL FAULT ?	A software fault has occurred.

2. Errors which terminate assembly of single statement only and are as follows:

Q Questionable syntax error.

T Truncation error.

* An assembler directive was encountered which is not valid in Z8002.

P A phase error occurred, i.e. a label's definition or value differed from first pass to second.

I An illegal character was encountered.

U An undefined symbol was encountered.

E No END directive was encountered.

L Statement length was greater than 92 characters; extra characters were ignored.

APPENDIX E

Target Tracking Process - Acceptance Tests and Alternate Routines.

This appendix describes the acceptance tests and the alternate routines for the following processes:

Read
Azimuth Inhibit
Range Inhibit
Set Binaries
Process Binaries
Approach/Recede Assessment
Coverage Assessment.

E.1.1. Read: Acceptance Test (see Fig.E.1.)

The following tests were carried out:

1. Check range gate within range, i.e. $1 \leqslant$ range gate $\leqslant 6$.
2. Check velocity gate within range, i.e. $1 \leqslant$ velocity gate $\leqslant 4$.
3. Check that range and velocity channel valid flags are set if 'target detected' flag is set.
4. Check that azimuth position counter within range, i.e. $0 \leqslant$ azimuth $\leqslant 29$.

E.1.2. Read: Alternate Routine (Fig.E.2.)

On failure of the primary read routine, the last azimuth position is read and updated. The 'target detected' flag is reset indicating no target.

E.2.1. Azimuth Inhibit: Acceptance Test (Fig.E.3.)

The following tests were carried out:

1. An error is signalled it 'target azimuth' is not valid and the flag 'within azimuth limits' is set.
2. Check that 'target azimuth' is within limits, i.e. $0 \leqslant$ target azimuth $\leqslant 29$.
3. Check that missing scans counter (for approach/recede assessment) is greater than or equal to zero. If less than zero for any reason then an error is flagged.

E.2.2. Azimuth Inhibit: Alternate Routine (Fig.E.4.)

The alternate routine for processing azimuth inhibit is based on a target detection decision. If no target is detected on the azimuth on which the alternate routine is entered, then all parameters are unmodified.

If a target is detected then 'target azimuth' is updated and the missing scans count is set to zero. In addition, the 'within azimuth limits' flag is cleared and cannot become set again until the radar has rotated 360^{o} minus half the width ot the azimuth inhibit arc. As a target has been detected then coverage information will be given, determined by a later process.

E.3.1. Range Inhibit: Acceptance Test (Fig.E.5.)

The following tests were carried out:

1. An error is indicated if 'azimuth inhibit' is set and 'range inhibit' is not set.

2. If 'target range' is valid and the missing scans count is larger than two, then an error is indicated if 'target range' does not equal the range gate set, or if 'range inhibit' is not set.

E.3.2. Range Inhibit: Alternate Routine (Fig.E.6.)

The alternate routine sets 'range inhibit' if 'azimuth inhibit' is set. If a target is detected and is not inhibited by azimuth considerations then 'target range' is updated.

In this simpler routine, range inhibition rules (i.e. ± 1 range gate) are not used. Thus if a target is detected following a system error (an error must have occurred in order to enter the alternate routine) then it is tracked. A target being tracked at the time of the error may be lost if multiple targets exist. It was thought better to track a target whose position is known exactly then use the position of a target whose characteristics may have been corrupted.

E.4.1. Set Binaries: Acceptance Test (Fig.E.7.)

The following tests were carried out:

1. Check that velocity binary is within limits, i.e. $1 \leqslant$ velocity binary $\leqslant 4$.

2. Check that range binary is within limits,

i.e. $1 \leqslant$ range binary $\leqslant 6$.

E.4.2. Set Binaries: Alternate Routine.

The alternate routine in this case is to re-execute the primary routine to set the appropriate binaries.

E.5.1. Process Binaries: Acceptance Test (Fig. E.8.)

The following tests were carried out:

1. If either alarm is set, ensure that 'binaries' flag is set.

2. If provisional external alarm is set, ensure that system is in search mode.

If the acceptance test passes and the provisional external alarm is set, then the external alarm is set.

E.5.2. Process Binaries: Alternate Routine.

The alternate routine would attempt a re-execution of the primary routine to determine whether the binaries are allowed to signal an alarm.

E.6.1. Approach/Recede Assessment: Acceptance Test (Fig. E.9.)

The following tests were carried out:

1. An error is indicated if both approach and recede are indicated.

2. An error is indicated if neither approach nor recede is indicated whilst the system is in track mode.

E.6.2. Approach/Recede Assessment: Alternate Routine (Fig. E.10.)

The alternate routine is a clean up and get out procedure, and is simply the setting of the approach/recede assessment to approach.

E.7.1. Coverage Assessment: Acceptance Test (Fig. E.11.)

The following tests were carried out:

1. If 'no coverage' flag is set, check that no provisional coverage indications are set.

2. Check that one and only one provisional coverage indication is set.

If the acceptance test passes, then set 'out of cover' or 'in cover' as appropriate.

E.7.2. Coverage Assessment: Alternate Routine (Fig.E.12.)

If either 'no coverage' or 'cancel' is set then coverage indications are set, otherwise a fail safe procedure is carried out by setting missile coverage to 'in cover'.

APPENDIX F

An Overview of MIL-STD 1553B

The 1553B standard was developed largely for aircraft internal transfers and defines a master slave communications protocol over a twisted pair. The exchange of messages along the twisted pair (bus) is precisely defined with ten allowable formats; the two formats which were used in this study are shown in Fig.F.1. Message formats can be divided into two groups, i.e. mode commands and data transfers. Mode commands are used to communicate with the bus hardware to aid the management of information flow, for example to shutdown a transmitter on a particular bus, as redundant buses are allowed. Data transfers along the bus consist of a message of not more than 32 words.

The standard allows three types of terminal to be connected to the bus. A terminal is defined within the standard as 'the electronic module necessary to interface the data bus with the subsystem and the subsystem with the data bus', while a subsystem is the combination of hardware and software required to perform a specific function. A master-slave protocol requires a master and is called a bus controller in the context of the 1553B standard. The bus controller is in charge of all communication over the bus, i.e. any message must be initiated by the bus controller. The second type of terminal is called a monitor; this terminal being assigned the task of receiving bus traffic and extracting selected information if required. A bus monitor is permitted to assume bus control if a set of predetermined bus transmission defects is detected. Finally a remote terminal is any terminal which is neither a bus controller nor a bus monitor.

Only three types of word are permitted with the standard. A word is a sequence of 16 bits plus sync (3 bit times) and parity (1 bit time) as shown in Fig.F.2. The first type of word is the command word which is always the first word of a message and is transmitted by the bus controller. The command word defines the type of message that will

follow. A transmit/receive bit within the command word establishes whether the message is to or from the remote terminal being addressed. A five bit address field specifies a unique address of a remote terminal for the purposes of the message. This address field allows a system to contain up to 31 remote terminals; the remaining address is used to communicate with all remote terminals. The second type of word is the status word, which is always the first word that is transmitted by a remote terminal in response to a message. This word contains the status condition of the remote terminal. The busy bit can be used by the remote terminal to indicate that it is unable to move data to or from the subsystem in compliance with the bus controllers command.
The message error bit indicates to the controller that one or more of the data words associated with the preceding receive command failed to pass the remote terminal's validity test. Finally a data word is used as part or whole of a message that may be up to 32 words in length.

The method of transmission along the bus is Manchester Two Bi-Phase level at a rate of 1.0 megabit per second. A logical '1' is transmitted as a positive pulse followed by a negative pulse, while a logical '0' is transmitted as a negative pulse followed by a positive pulse. A transition through zero occurs at the midpoint of each bit time as shown in Fig.F.3.

A 1553B word is valid if it conforms to the following criteria:

1. The word begins with a valid sync field.
2. The bits are in a valid Manchester Two Bi-Phase level code.
3. The information field has 16 bits plus parity.
4. The word parity is odd.

APPENDIX G

Worst Case Limits for Parallel Realisation of Digital Controller.

The parallel realisation of the digital controller results in four parallel units which are added to give a guidance demand. The acceptance test for each of these four units was based on worst case outputs of the units. The worst case value for the guidance demand was achieved by the addition of the worst case values for the units.

The worst case outputs were obtained by using an input which is equivalent to a 90° step. A simulation run was then carried out on the PDP 11 and the following results were obtained.

	Worst Case Output	Value used in Acceptance test
Unit 1	± 5.27	± 6
Unit 2	± 0.814	± 1
Unit 3	± 24.8	± 25
Unit 4	± 4.5	± 5
Guidance Demand		± 37

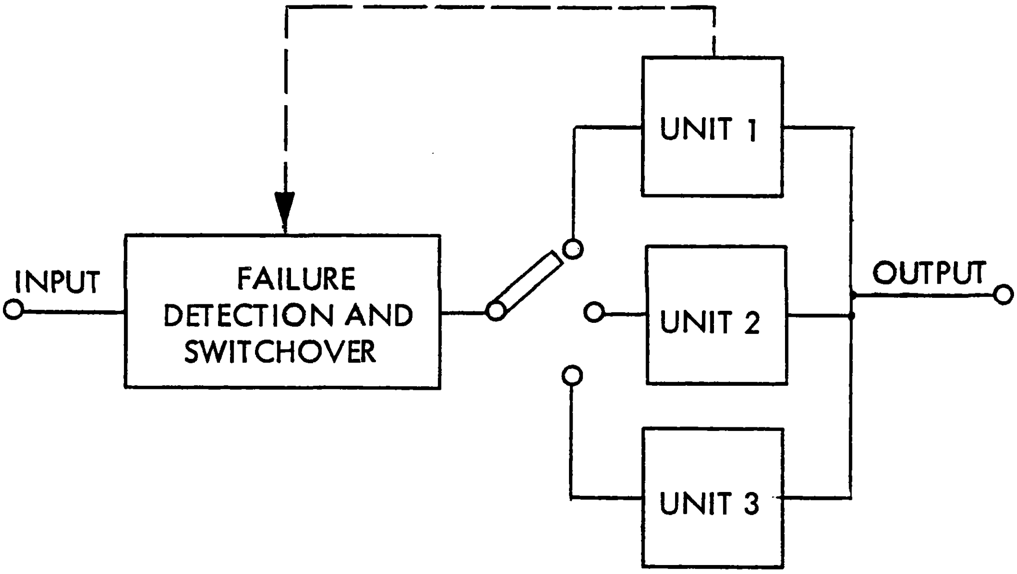


Fig.2.1. Cold Standby Redundancy.

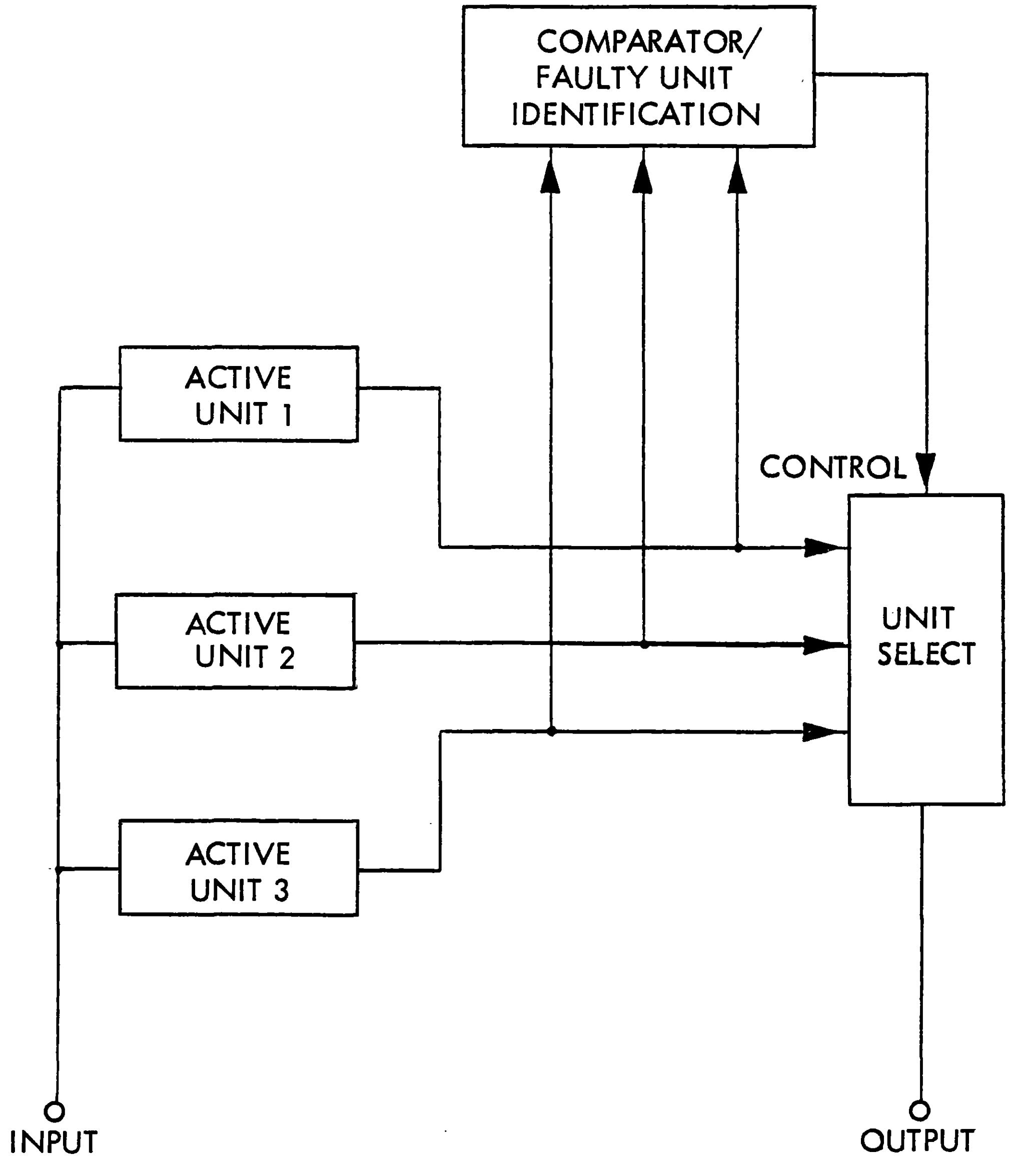


Fig.2.2. Hot Standby Redundancy.

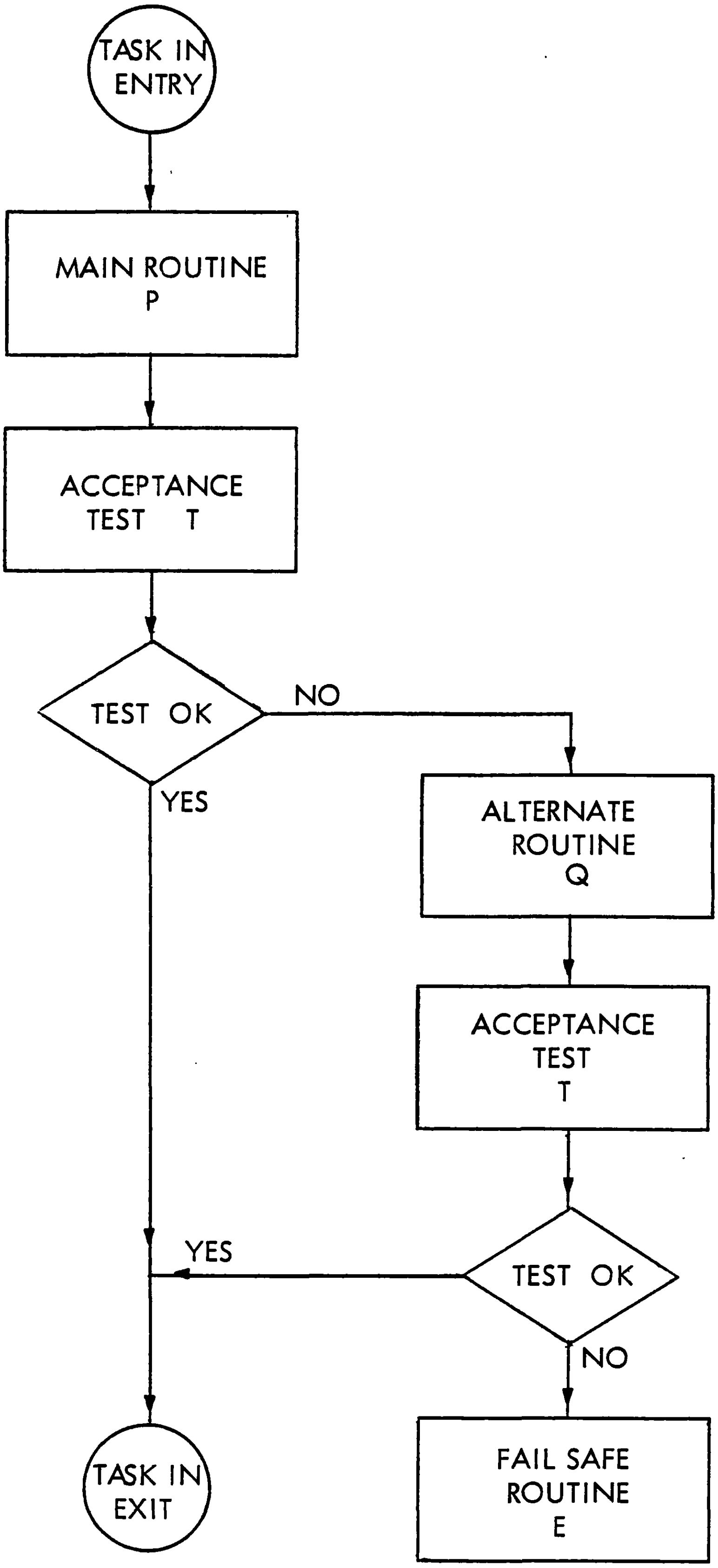


Fig.2.3. The Recovery Block.

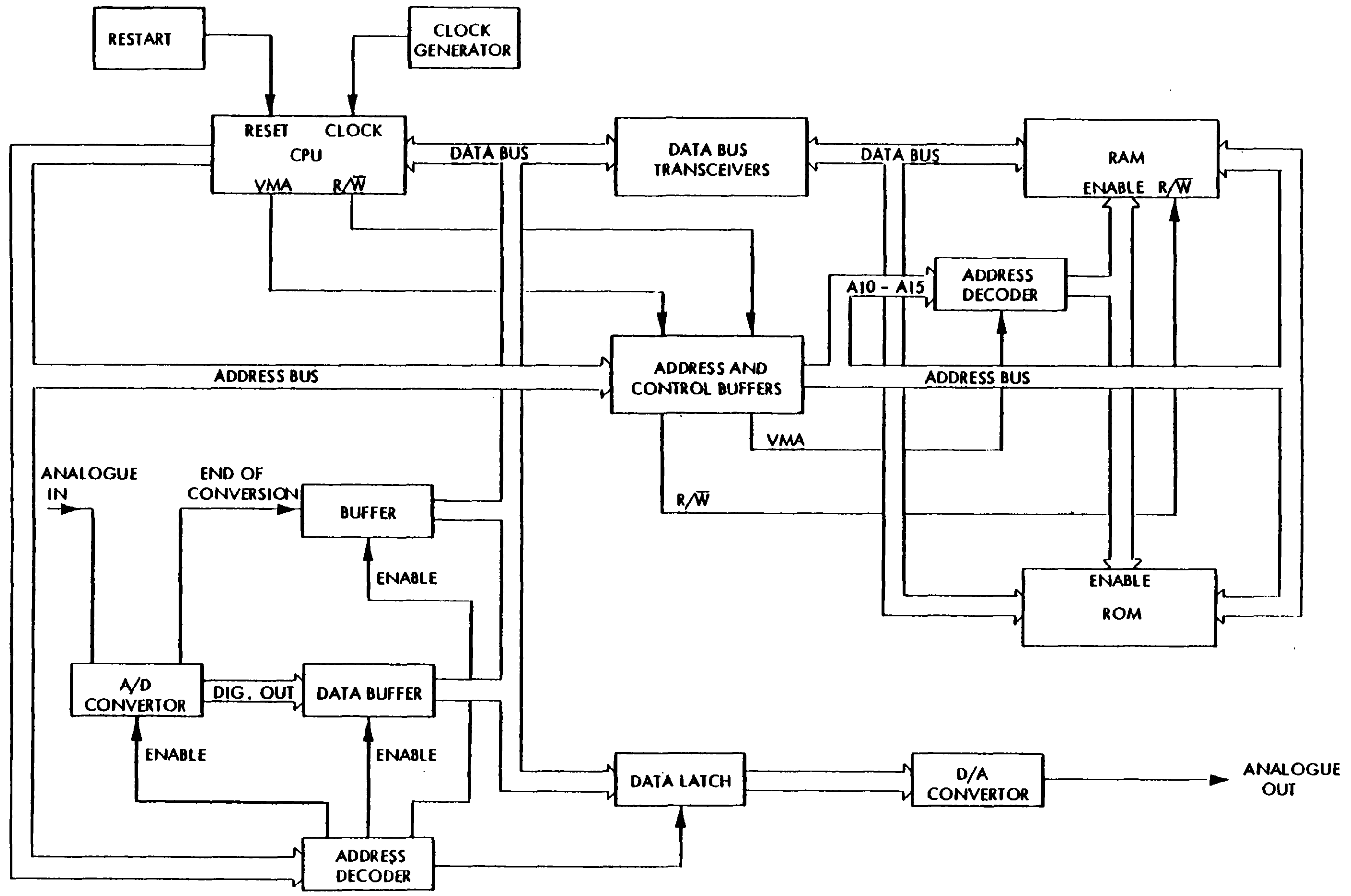


Fig.3.1. A Typical Microprocessor System.

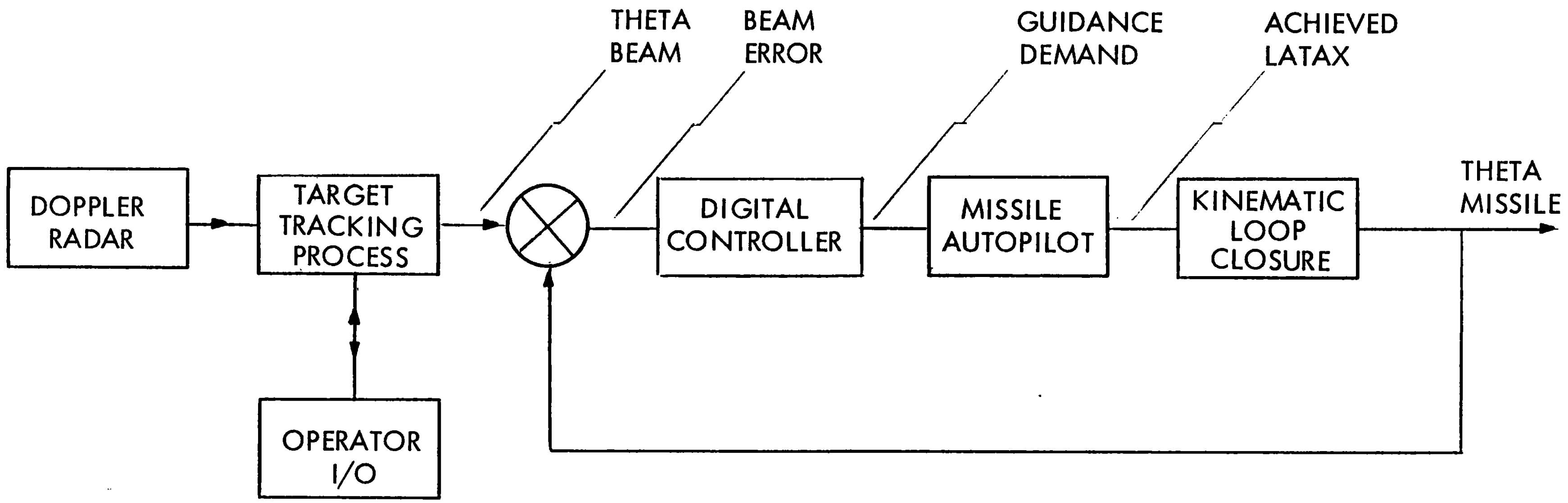


Fig.4.1. Real Time System Schematic.

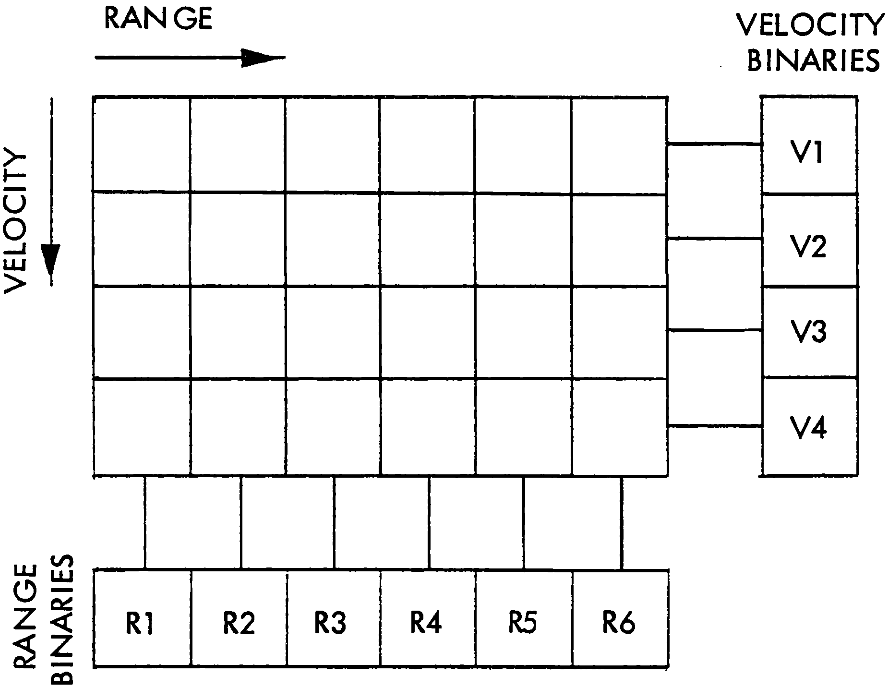


Fig.4.2. Range/Velocity Gate Matrix.

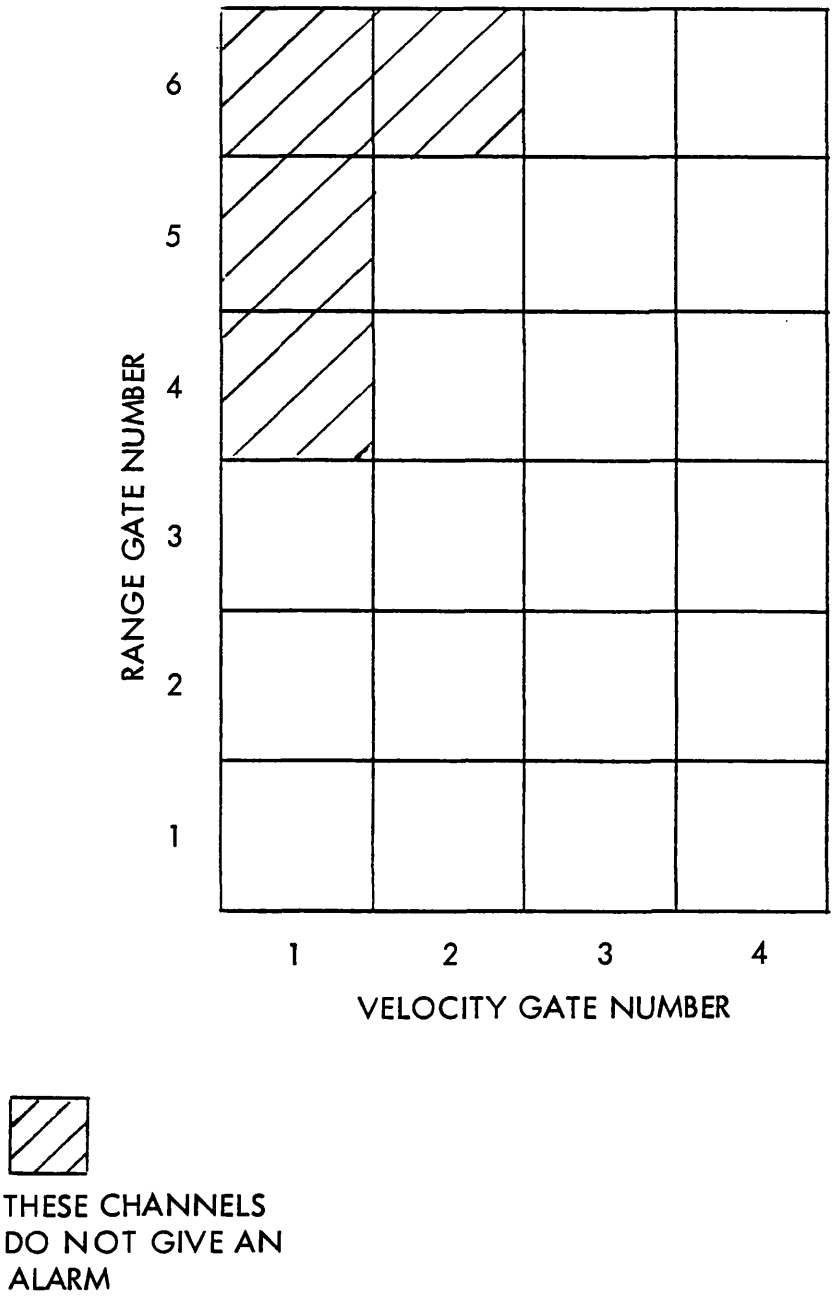
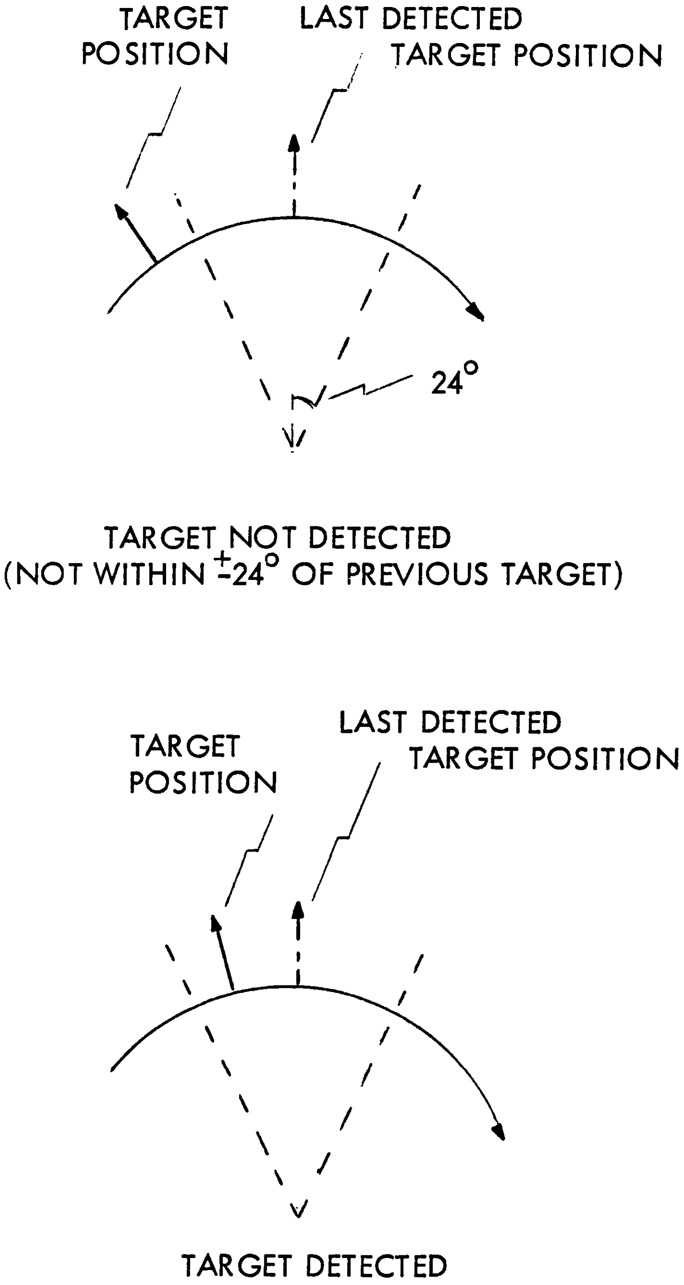


Fig.4.3. Taboo Channels.



TARGET NOT DETECTED
(NOT WITHIN ±24° OF PREVIOUS TARGET)

TARGET DETECTED

Fig.4.4. Principle of Azimuth Inhibit.

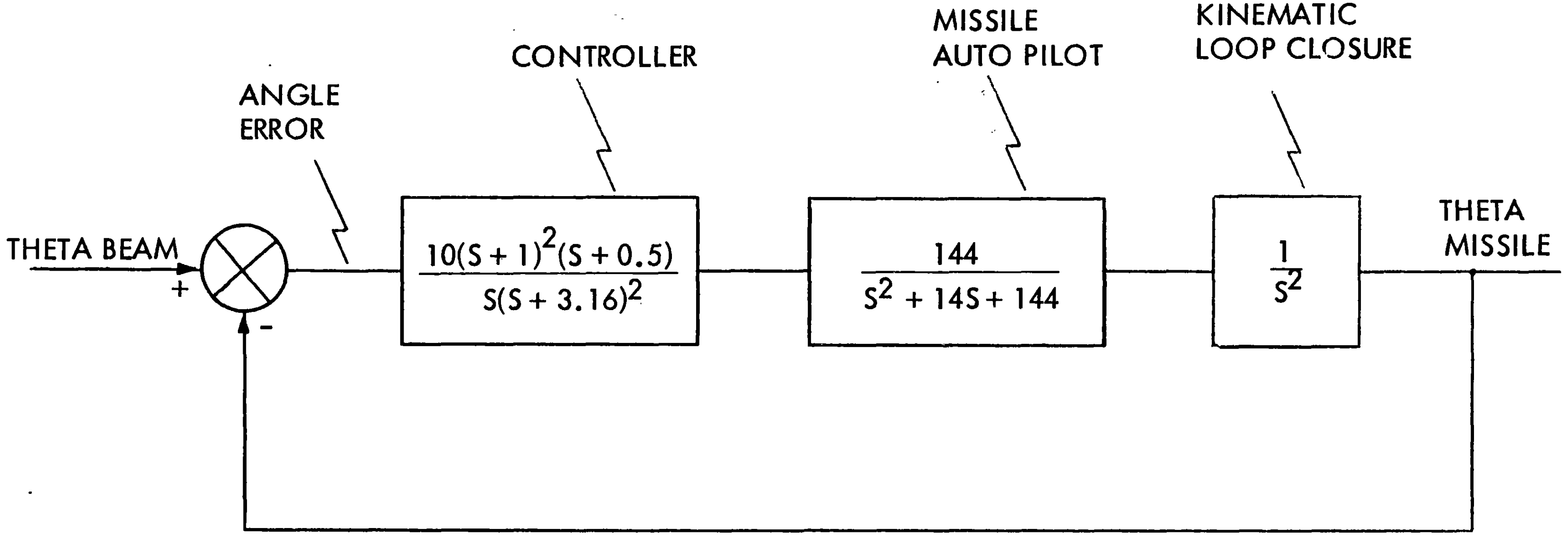


Fig.4.5. Missile Guidance Loop.

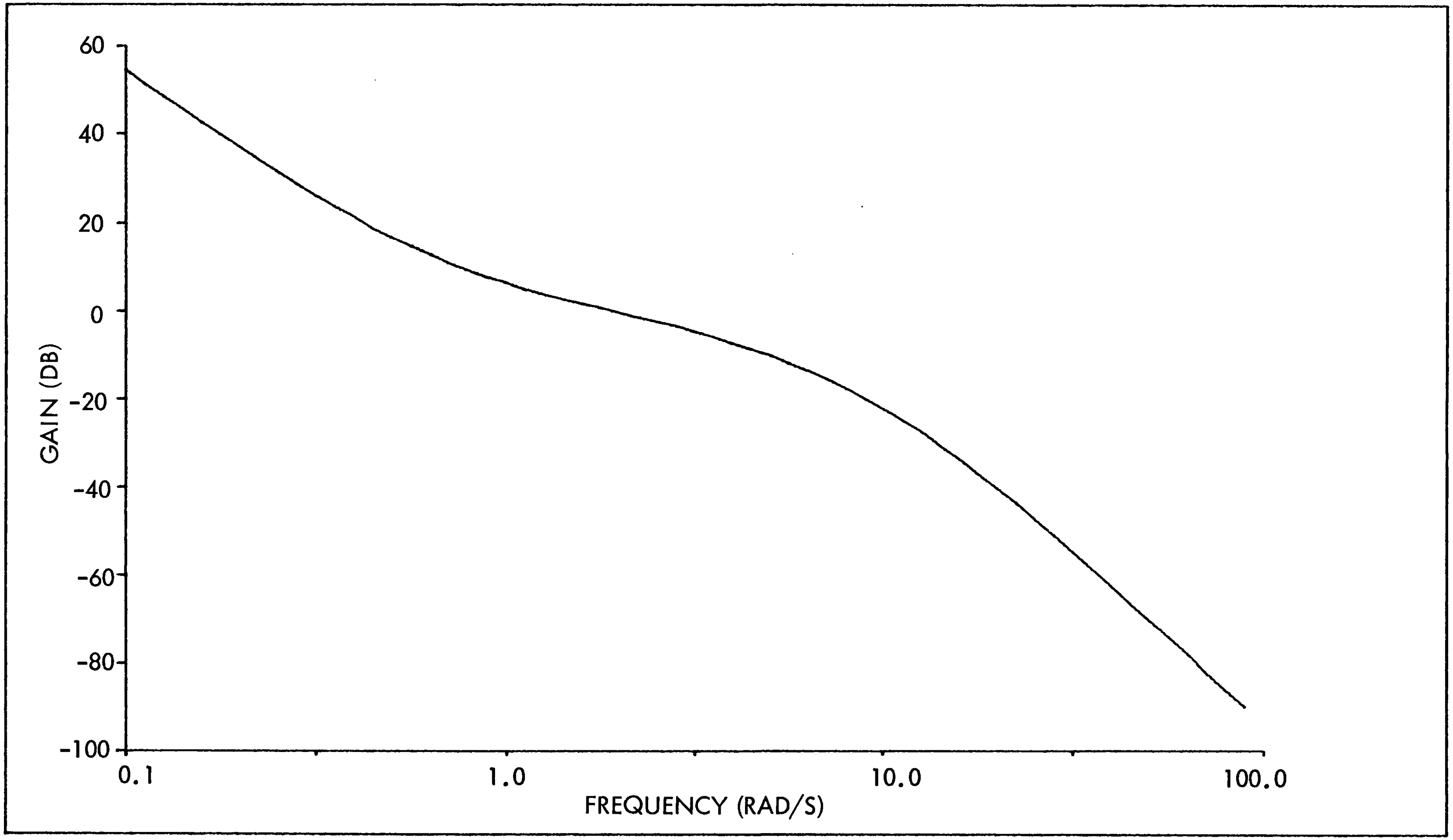


Fig.4.6.a. Gain Plot of Missile Guidance Loop.

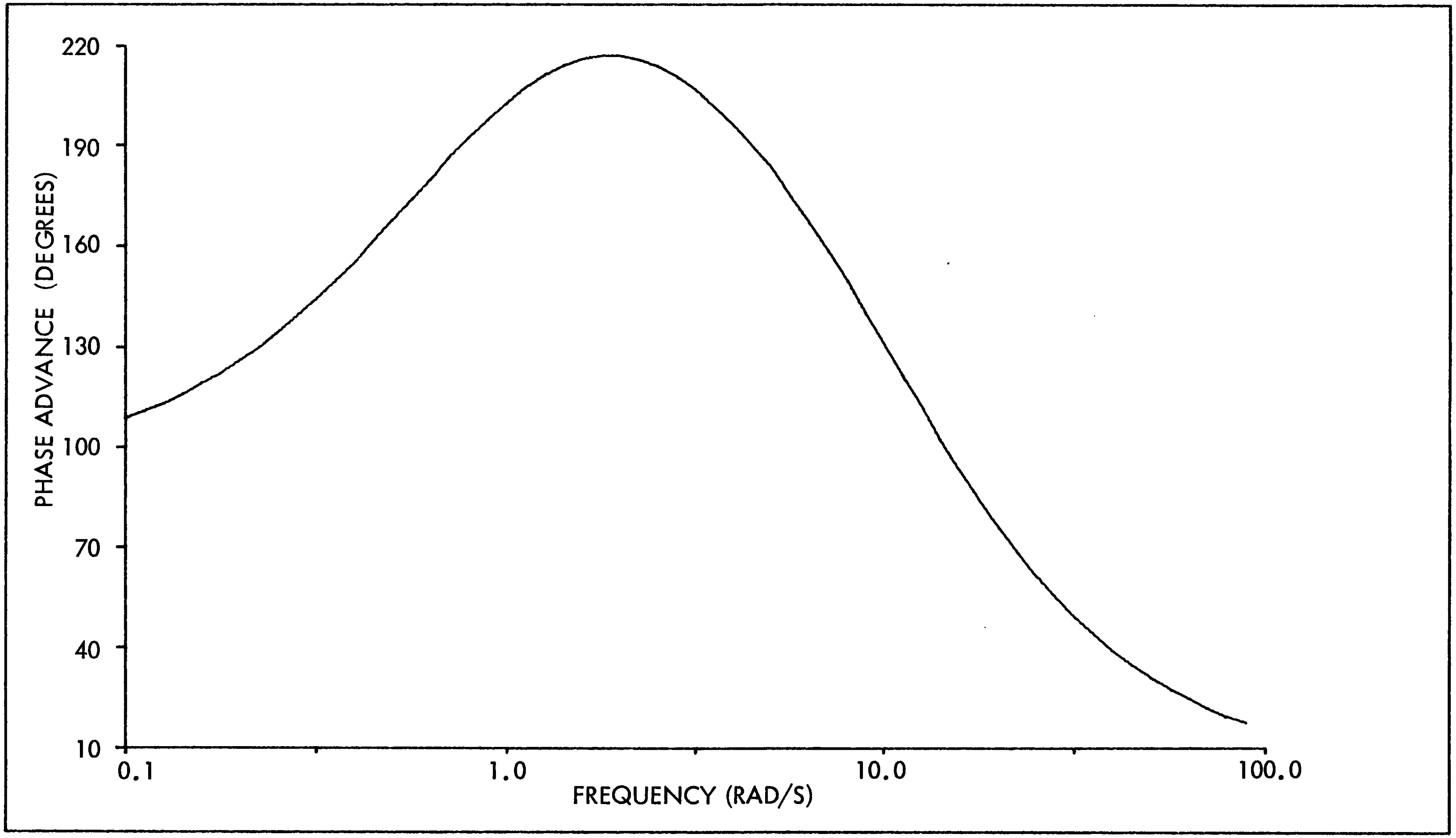


Fig.4.6.b. Phase Plot of Missile Guidance Loop.

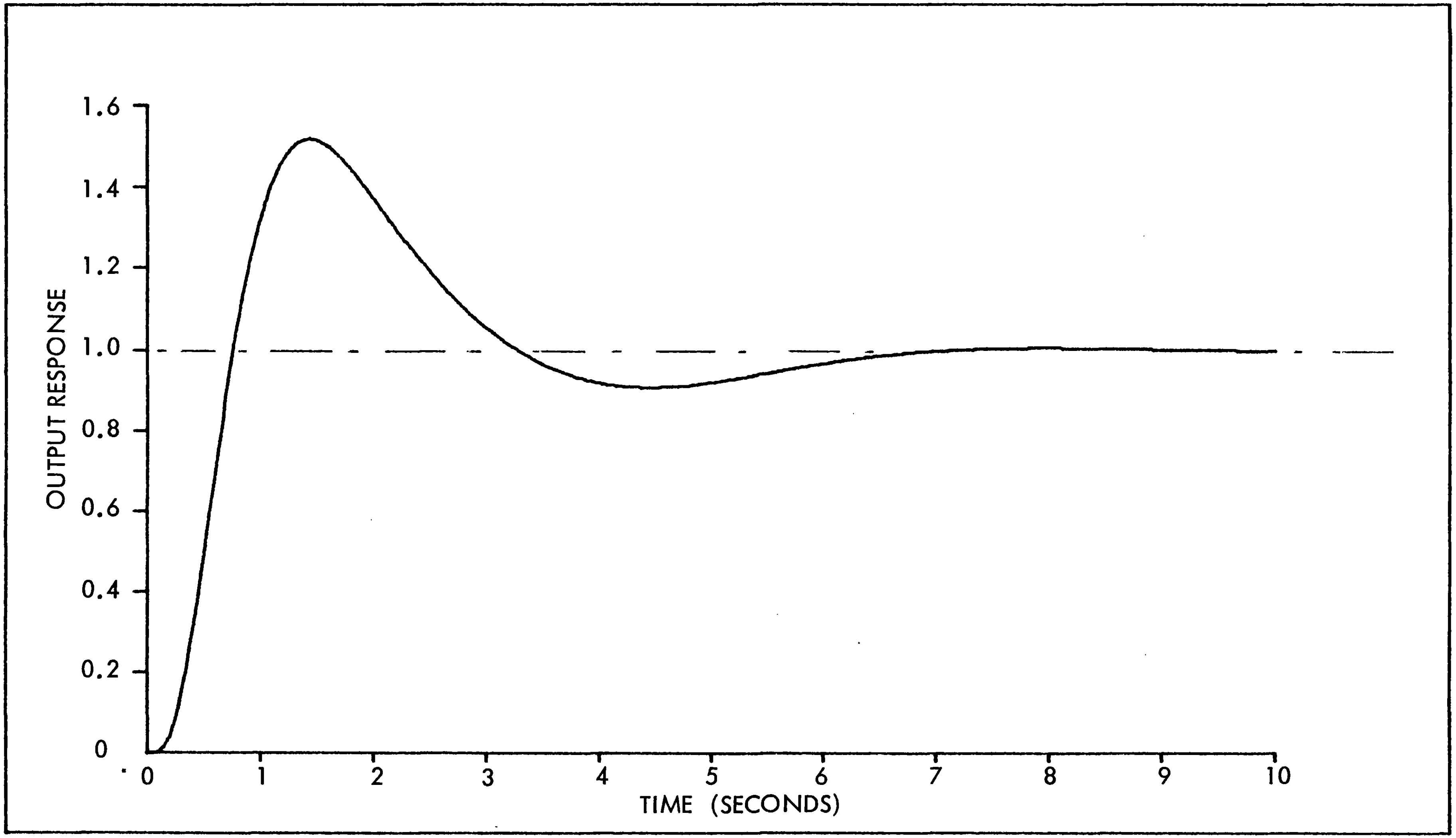


Fig.4.7. Step Response of Missile Guidance Loop.

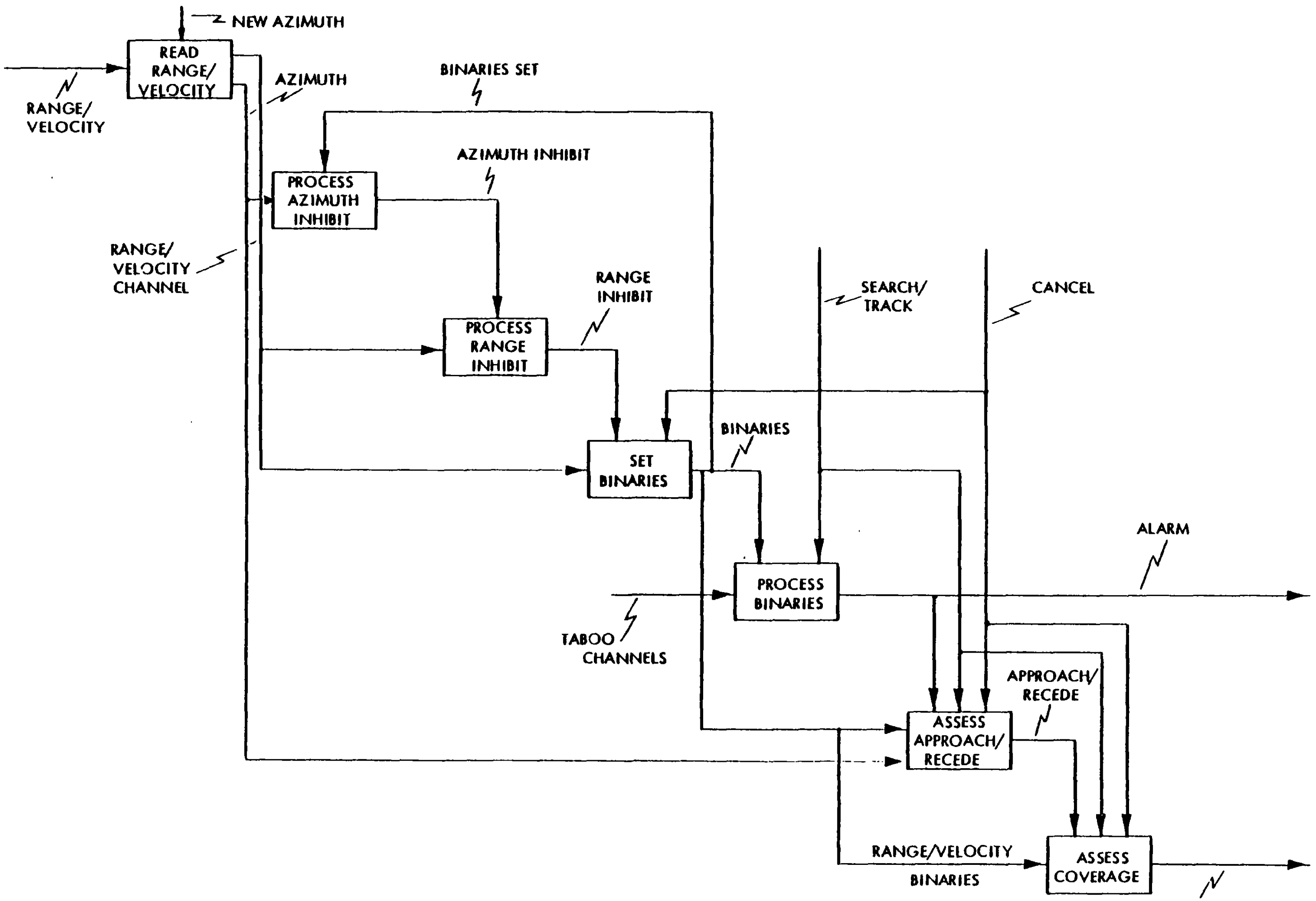


Fig. 5.1. Target Tracking Process.

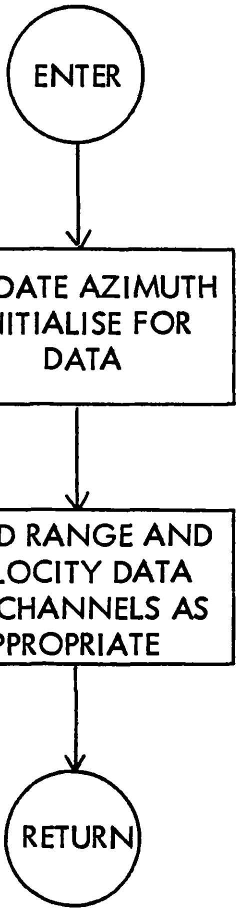

ENTER
UPDATE AZIMUTH INITIALISE FOR DATA
READ RANGE AND VELOCITY DATA SET CHANNELS AS APPROPRIATE
RETURN

Fig.5.2. Read Routine.

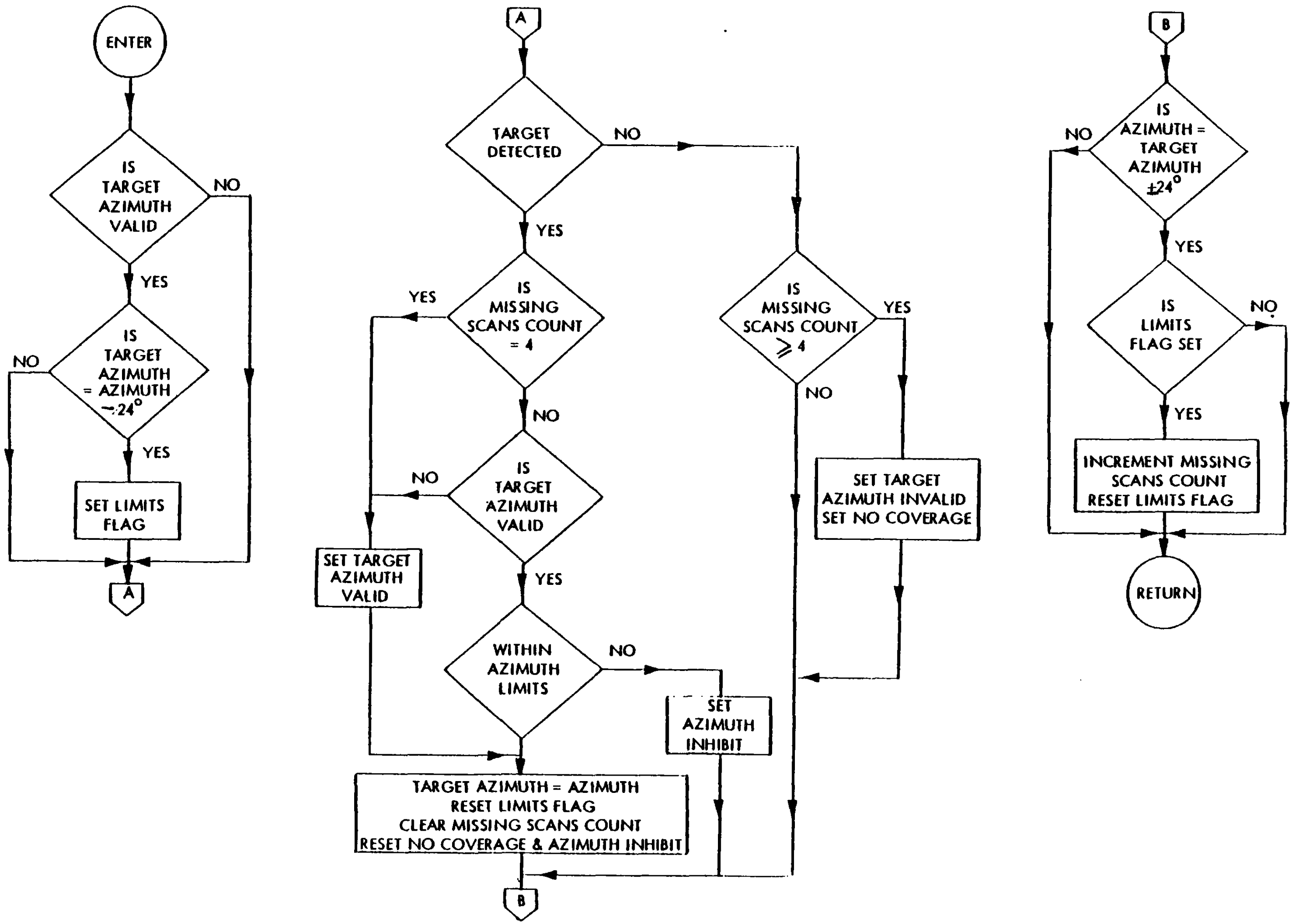

ENTER
IS TARGET AZIMUTH VALID
NO
YES
IS TARGET AZIMUTH = AZIMUTH ±24°
NO
YES
SET LIMITS FLAG
A
A
TARGET DETECTED
NO
YES
IS MISSING SCANS COUNT = 4
YES
NO
IS TARGET AZIMUTH VALID
NO
YES
SET TARGET AZIMUTH VALID
WITHIN AZIMUTH LIMITS
NO
SET AZIMUTH INHIBIT
TARGET AZIMUTH = AZIMUTH
RESET LIMITS FLAG
CLEAR MISSING SCANS COUNT
RESET NO COVERAGE & AZIMUTH INHIBIT
IS MISSING SCANS COUNT ⩾ 4
YES
NO
SET TARGET AZIMUTH INVALID SET NO COVERAGE
B
B
IS AZIMUTH = TARGET AZIMUTH ±24°
NO
YES
IS LIMITS FLAG SET
NO
YES
INCREMENT MISSING SCANS COUNT RESET LIMITS FLAG
RETURN

Fig. 5.3. Process Azimuth Inhibit.

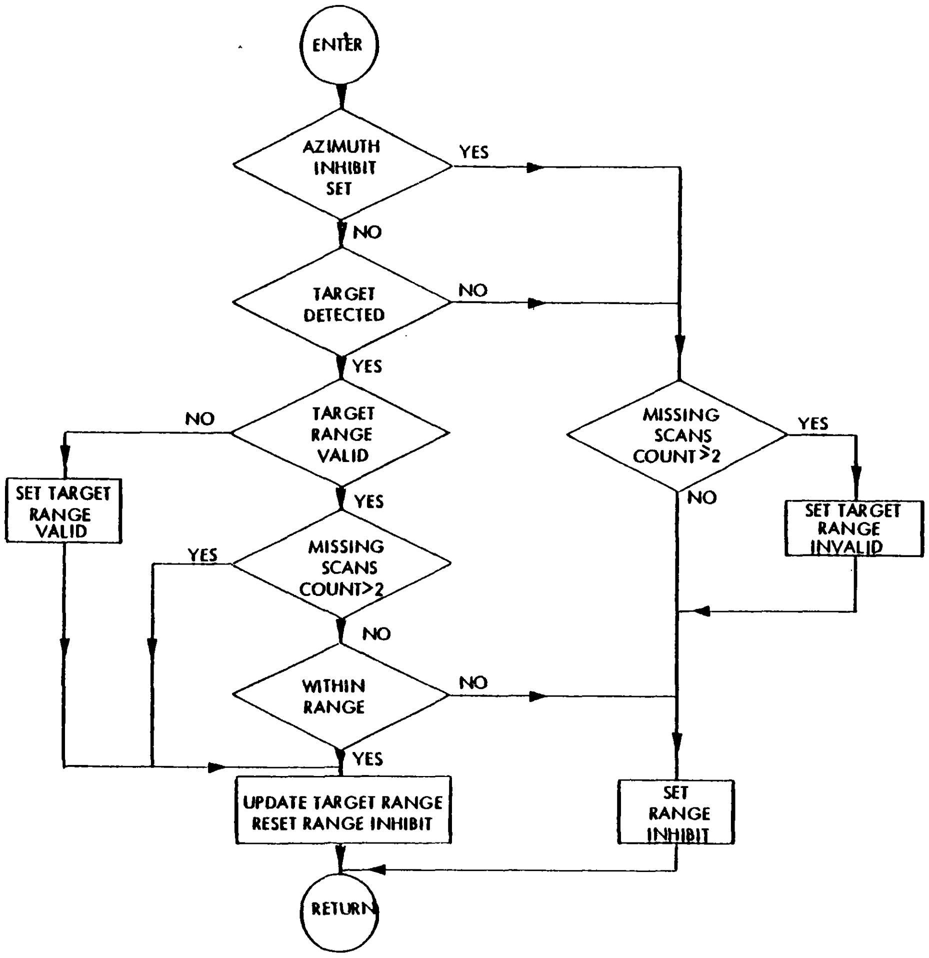


Fig. 5.4. Process Range Inhibit.

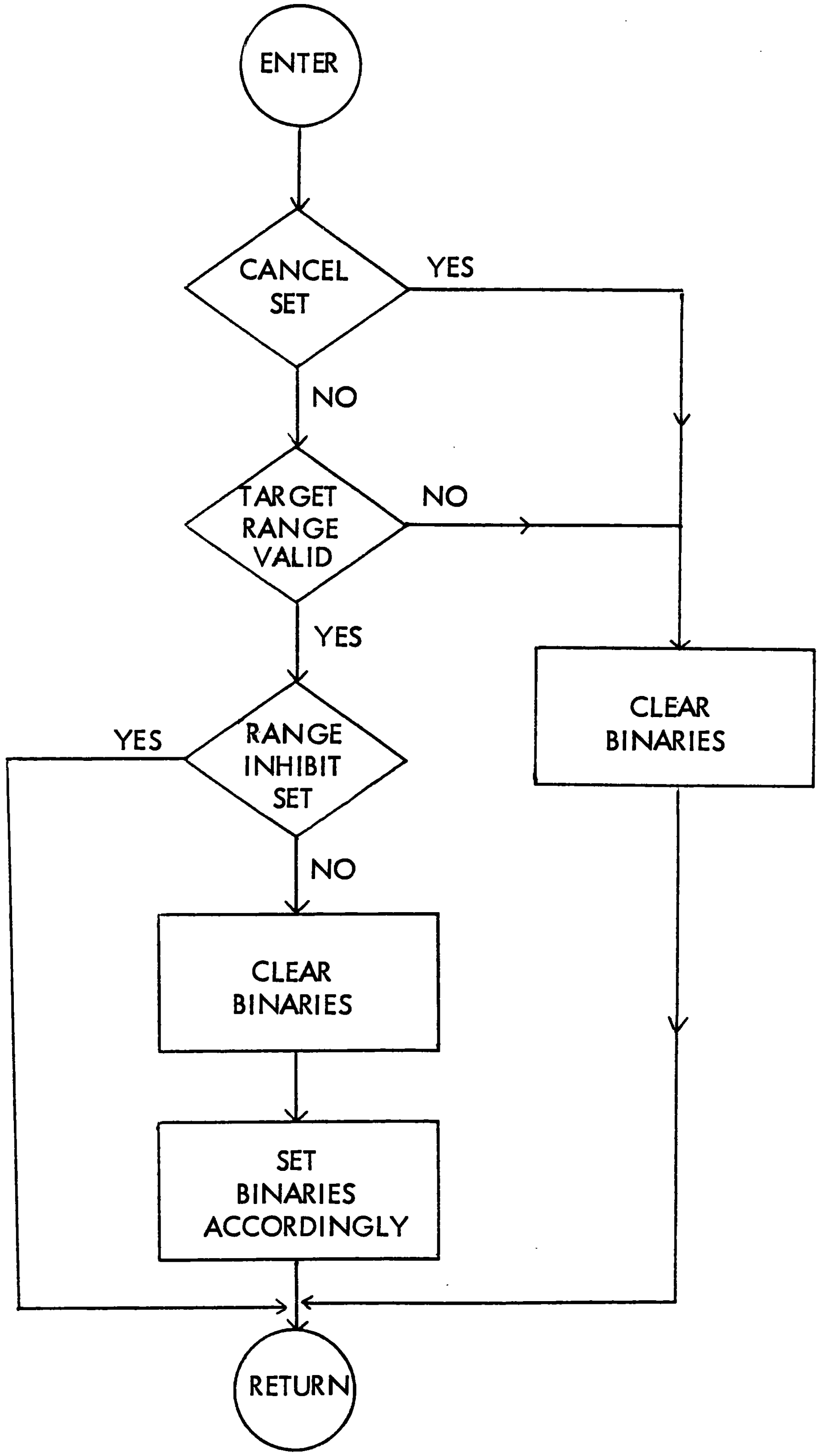


Fig.5.5. Set Binaries.

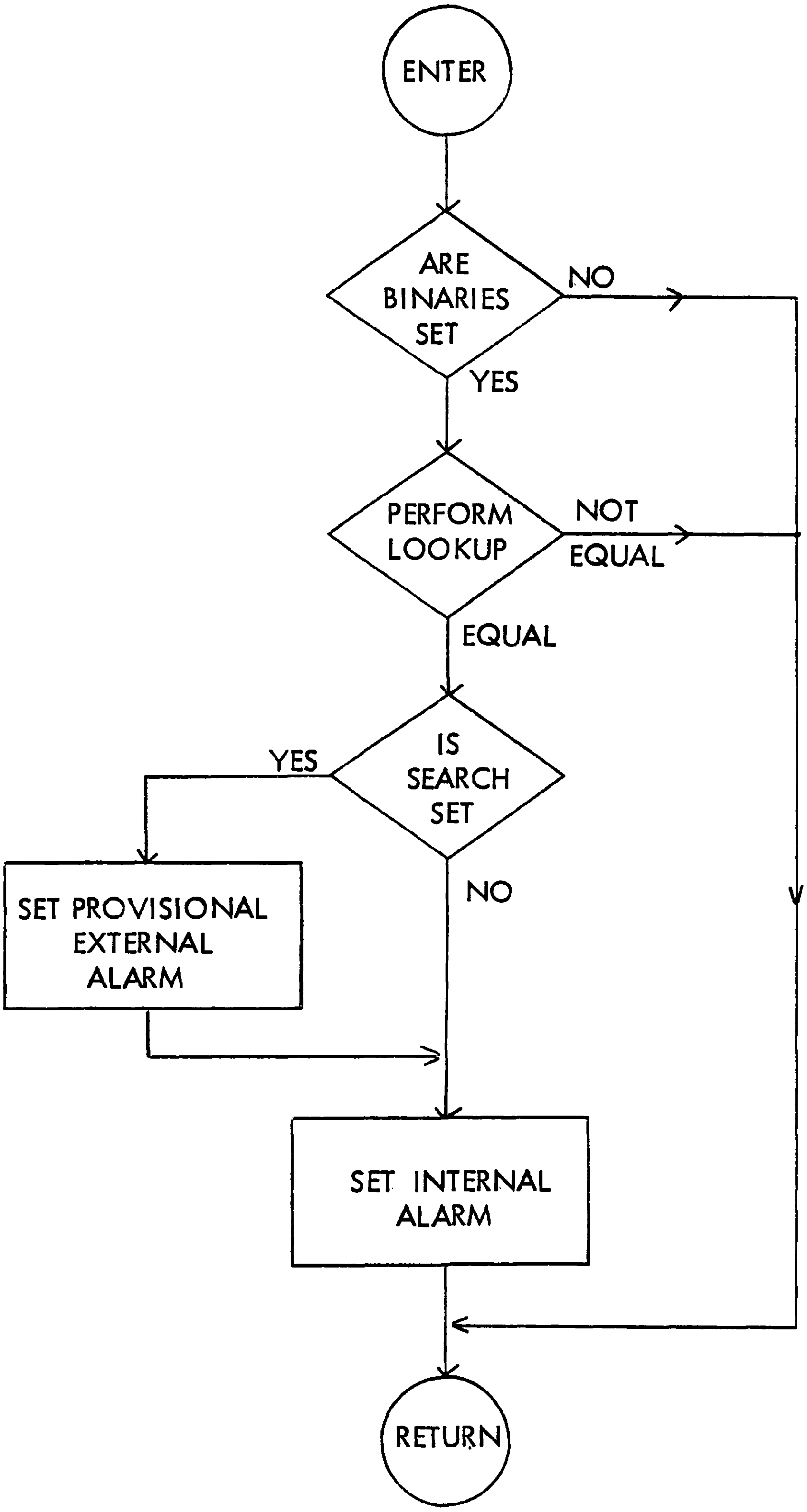


Fig.5.6. Process Binaries.

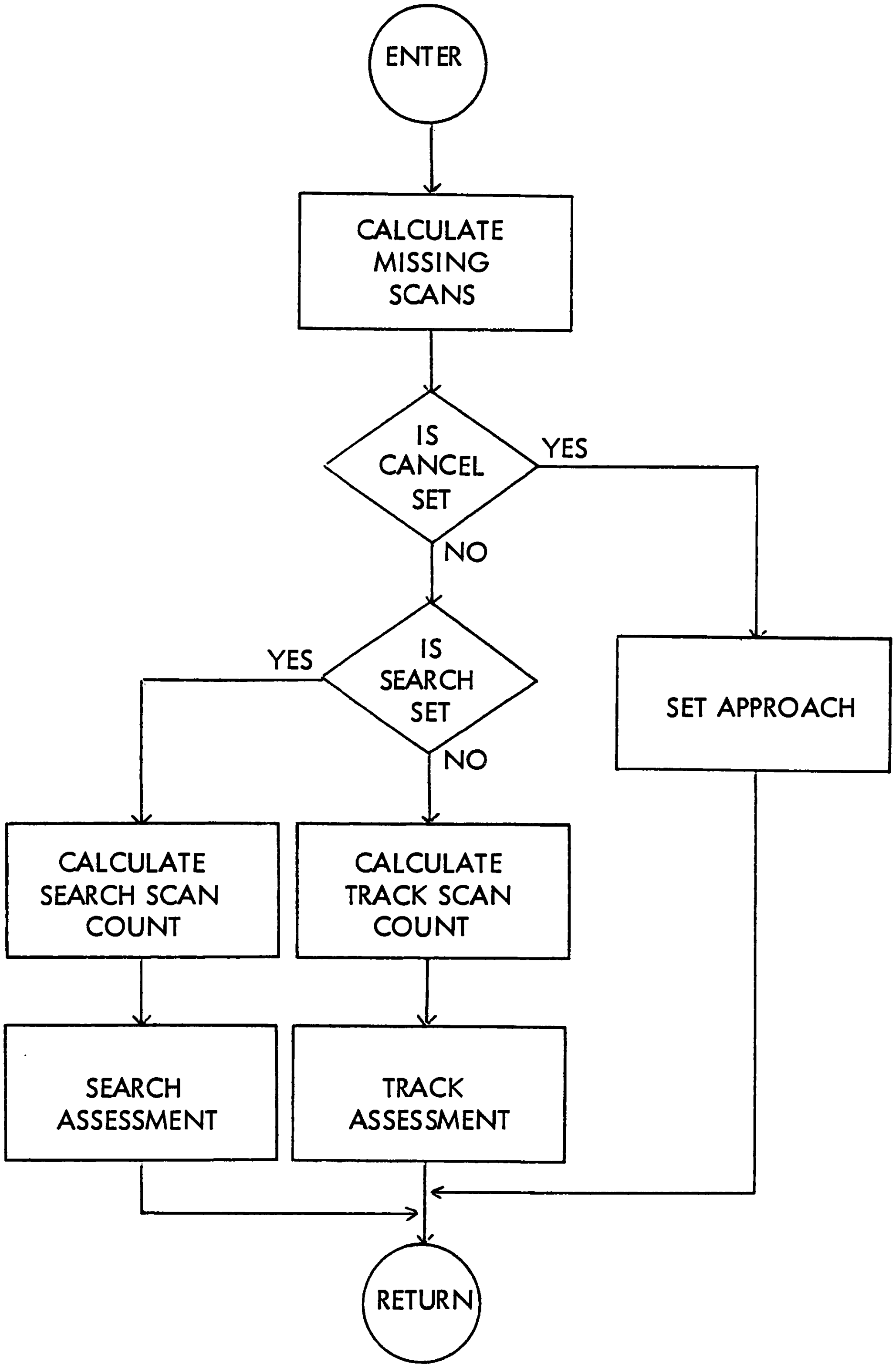


Fig.5.7. Approach/Recede Assessment.

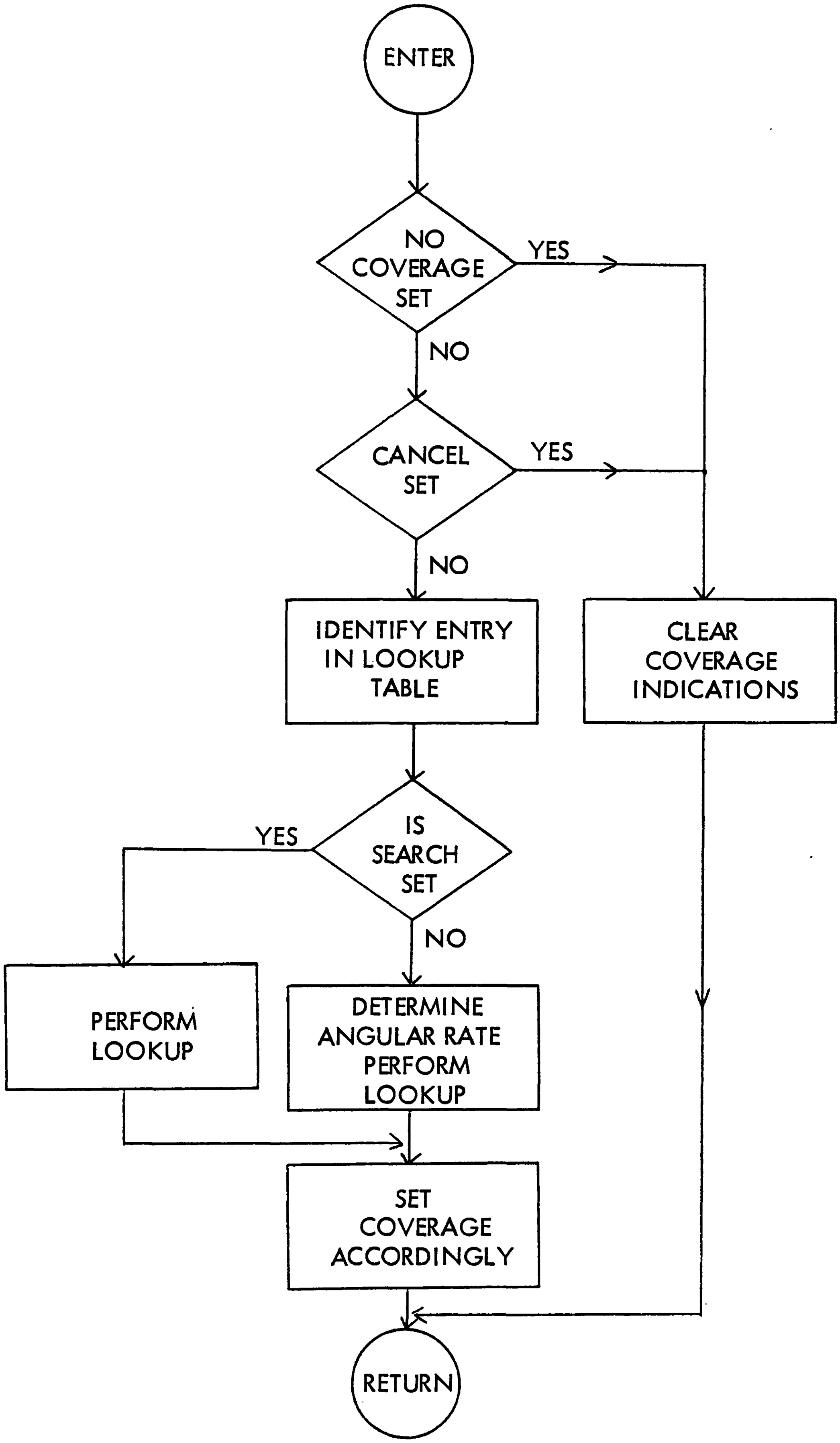


Fig.5.8. Coverage Assessment.

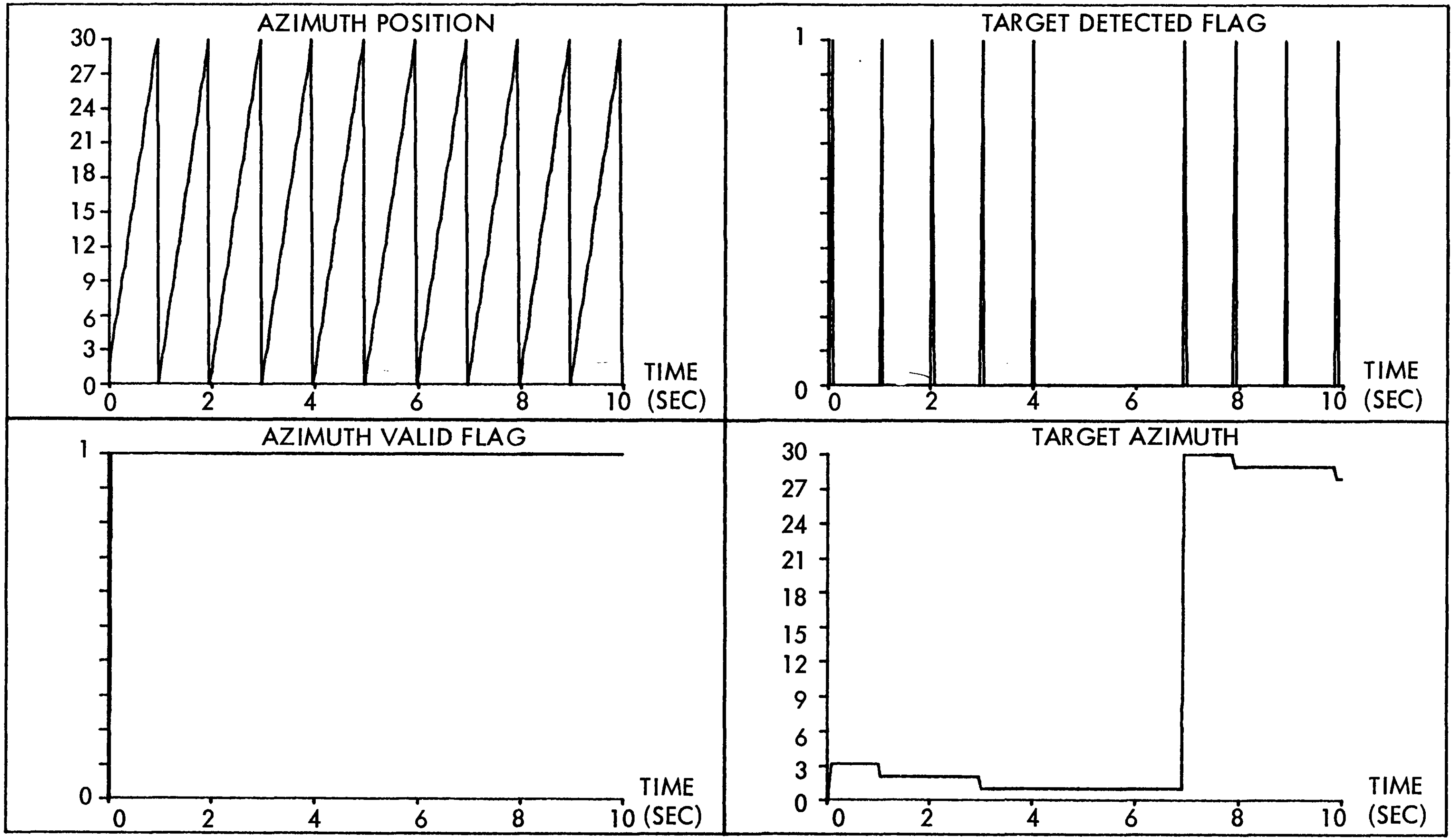


Fig.5.9.a. Targe t Tracking Process Outputs.

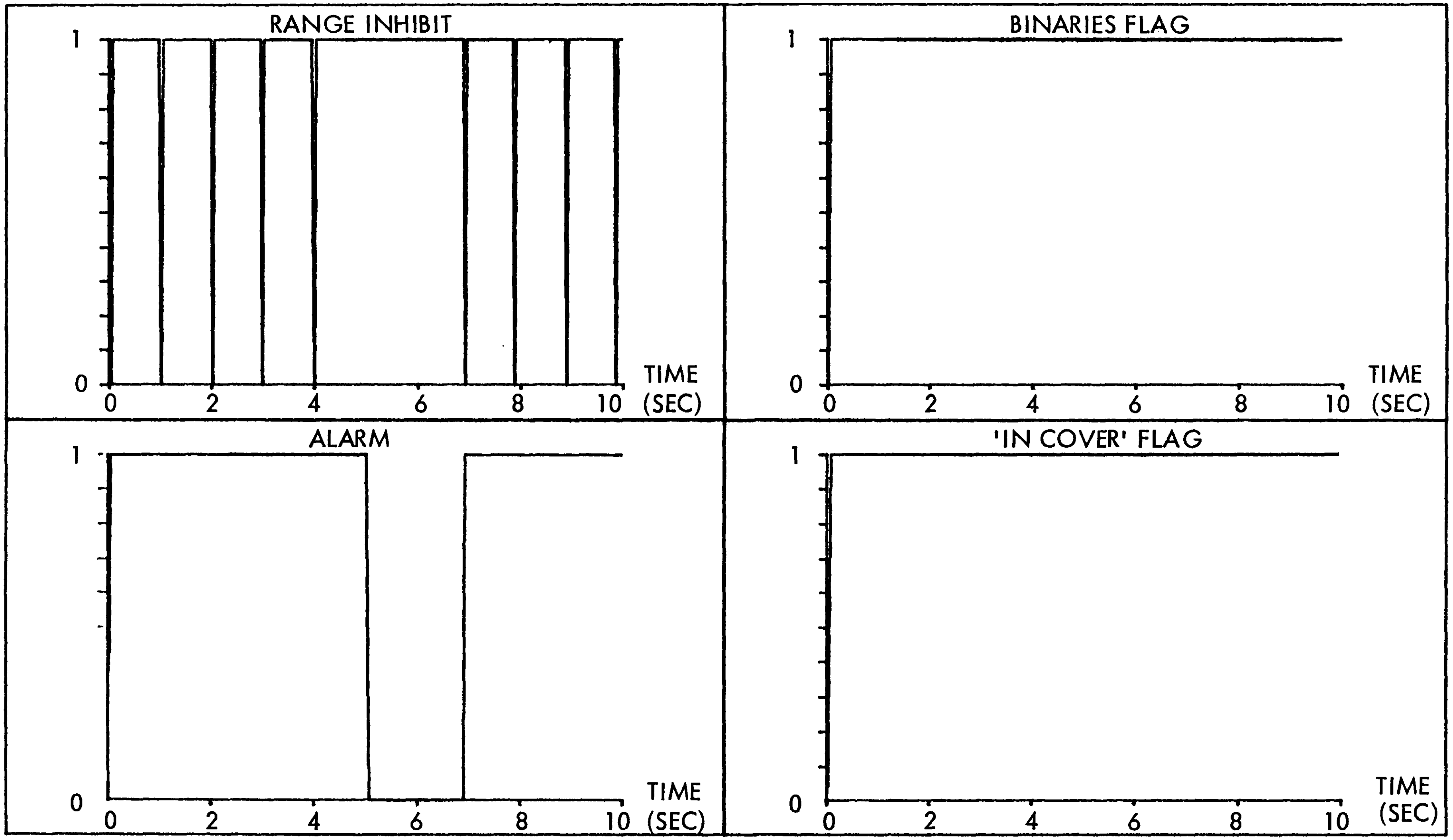


Fig.5.9.b. Target Tracking Process Outputs.

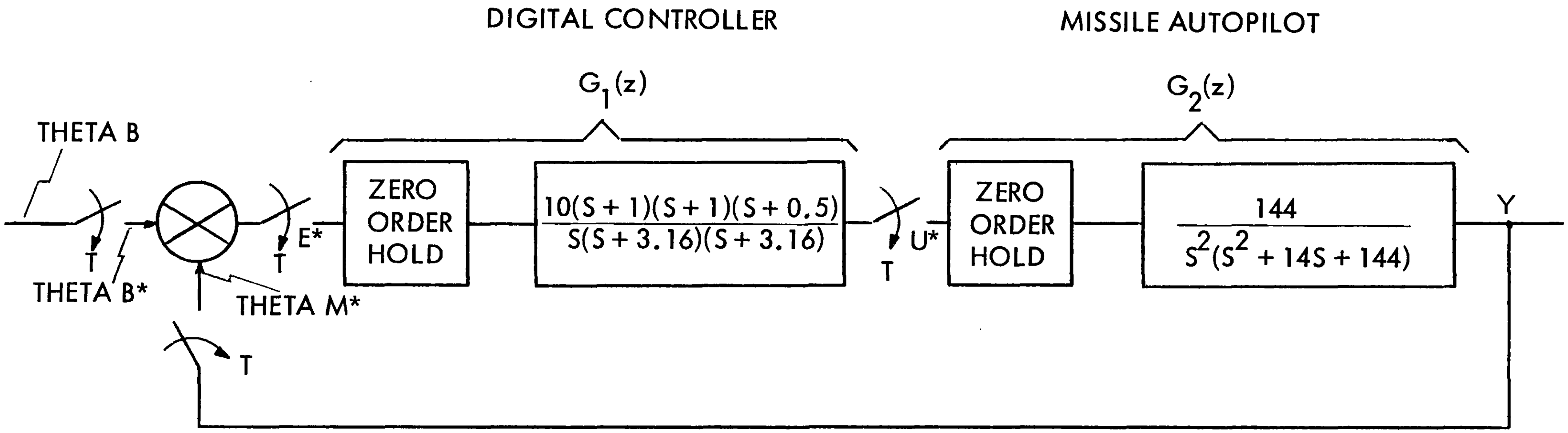


Fig.5.10. Digitisation of Guidance Loop.

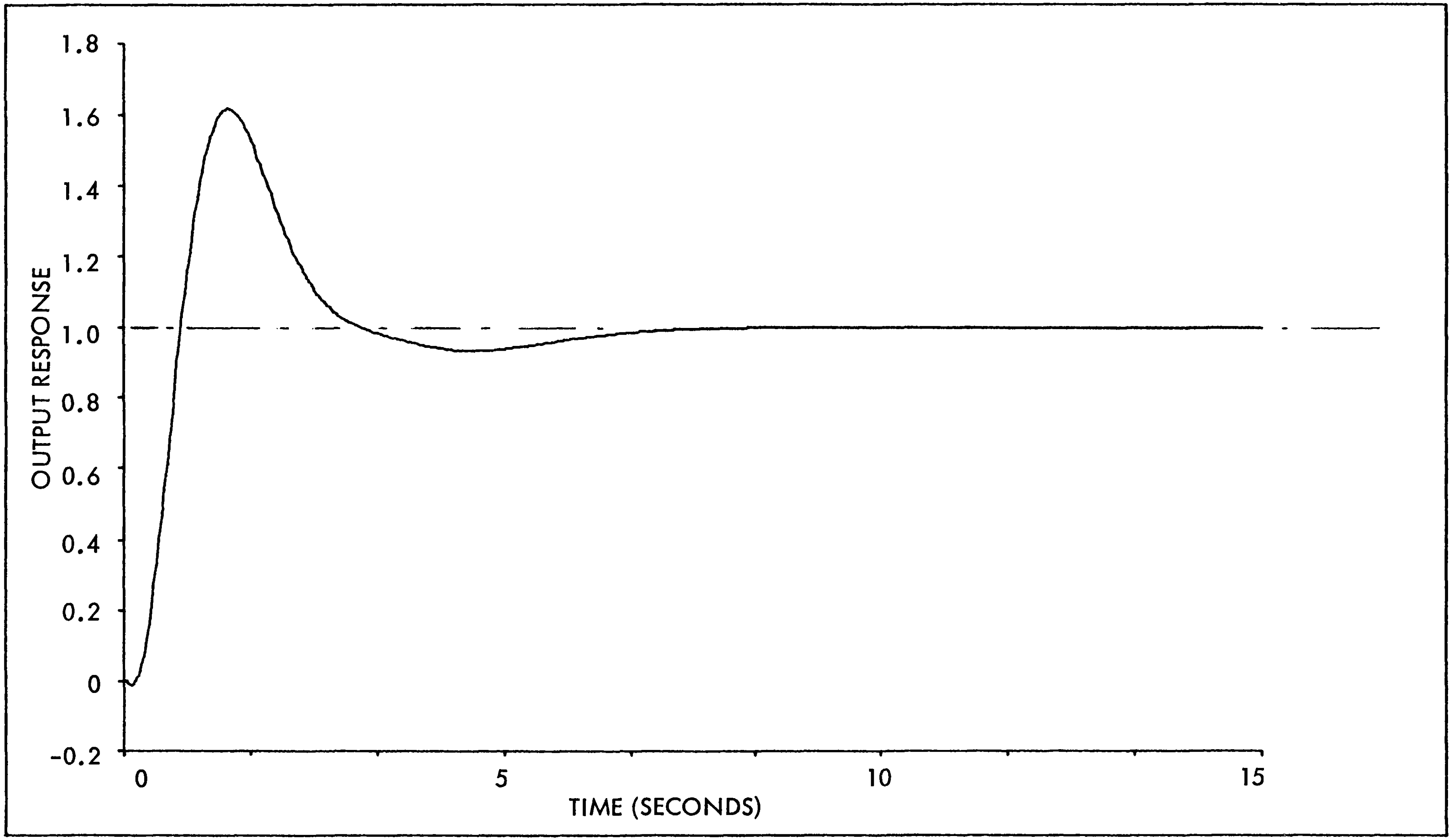


Fig.5.11. Unit Step Response for Direct Realisation.

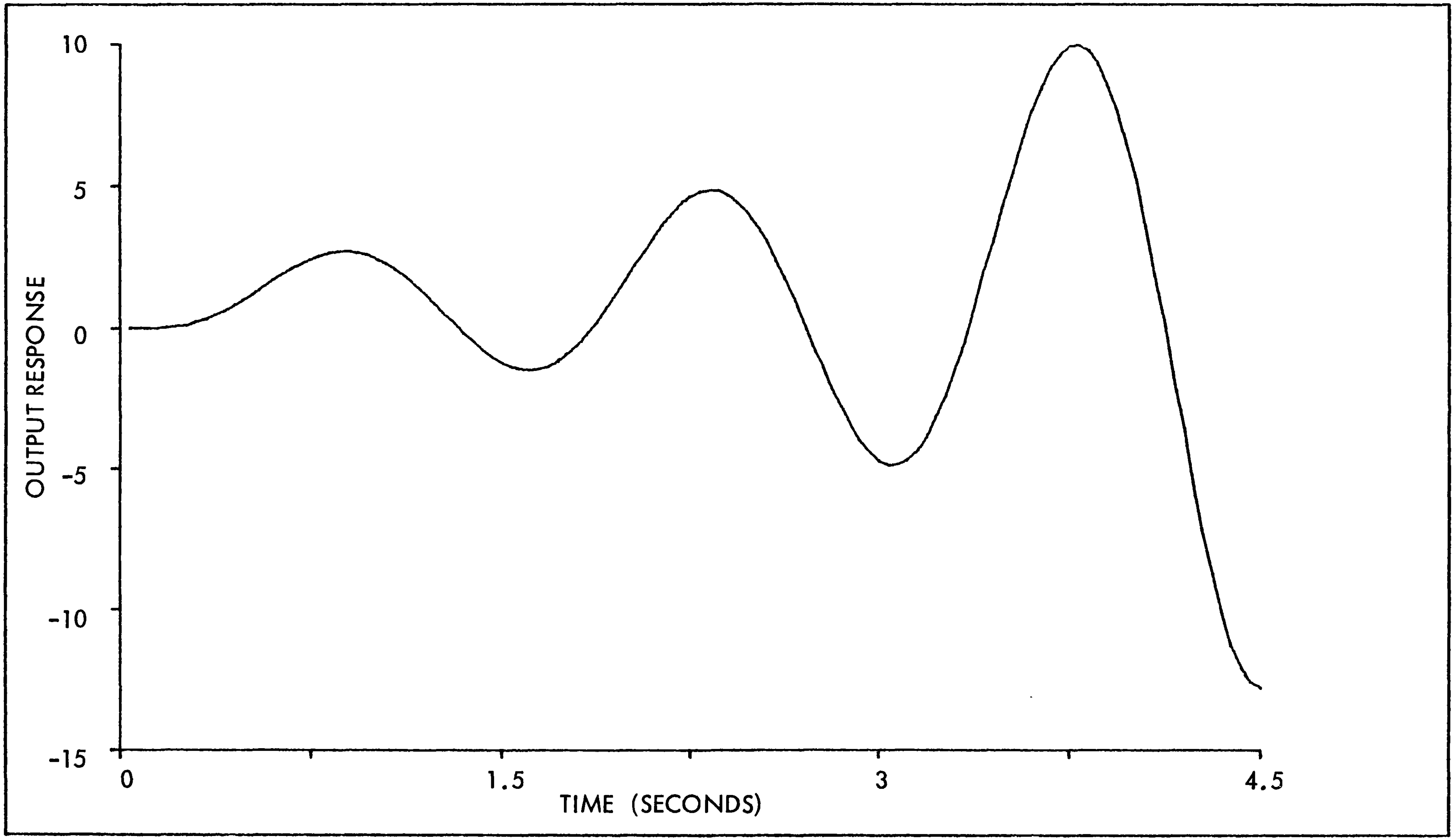


Fig.5.12. Unit Step Response for Direct Realisation: Binary Rounded Coefficients.

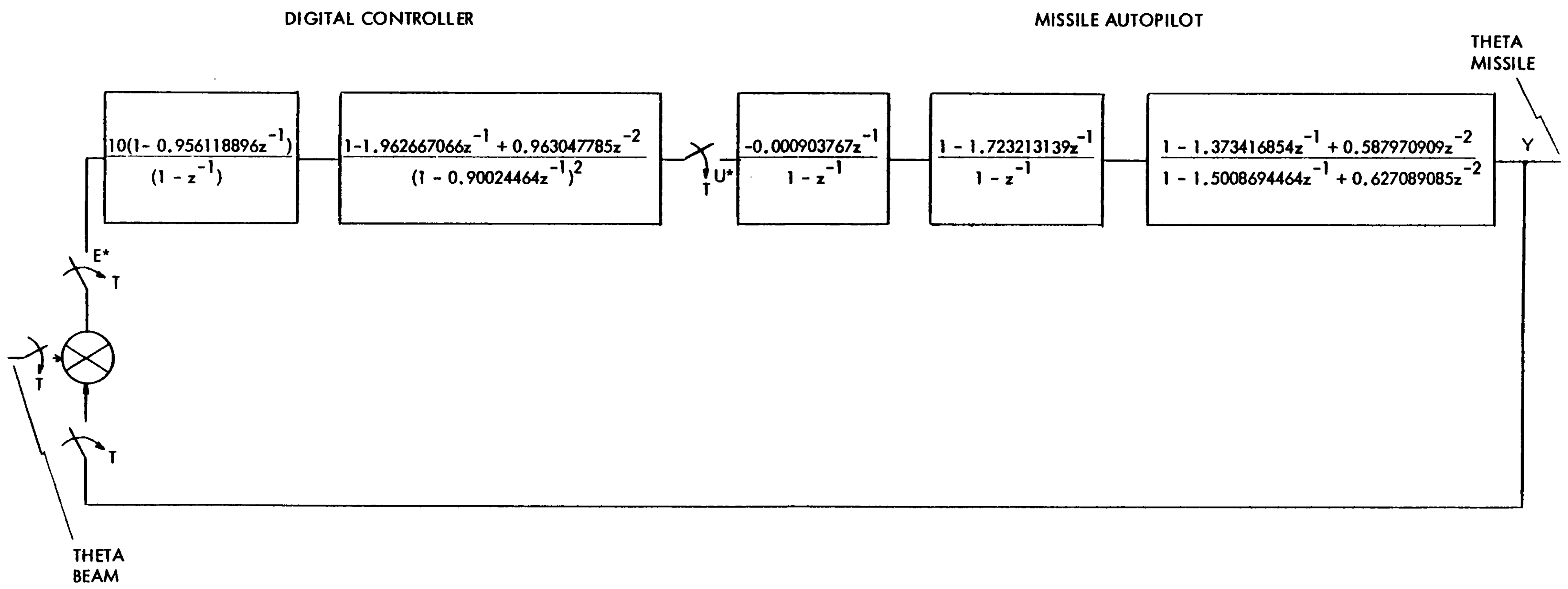

DIGITAL CONTROLLER
MISSILE AUTOPILOT
THETA MISSILE
Y
$\frac{10(1-0.956118896z^{-1})}{(1-z^{-1})}$
$\frac{1-1.962667066z^{-1}+0.963047785z^{-2}}{(1-0.90024464z^{-1})^2}$
U^*_T
$\frac{-0.000903767z^{-1}}{1-z^{-1}}$
$\frac{1-1.723213139z^{-1}}{1-z^{-1}}$
$\frac{1-1.373416854z^{-1}+0.587970909z^{-2}}{1-1.5008694464z^{-1}+0.627089085z^{-2}}$
E*
T
T
T
THETA BEAM

Fig.5.13. Cascade Realisation of Missile Guidance Loop.

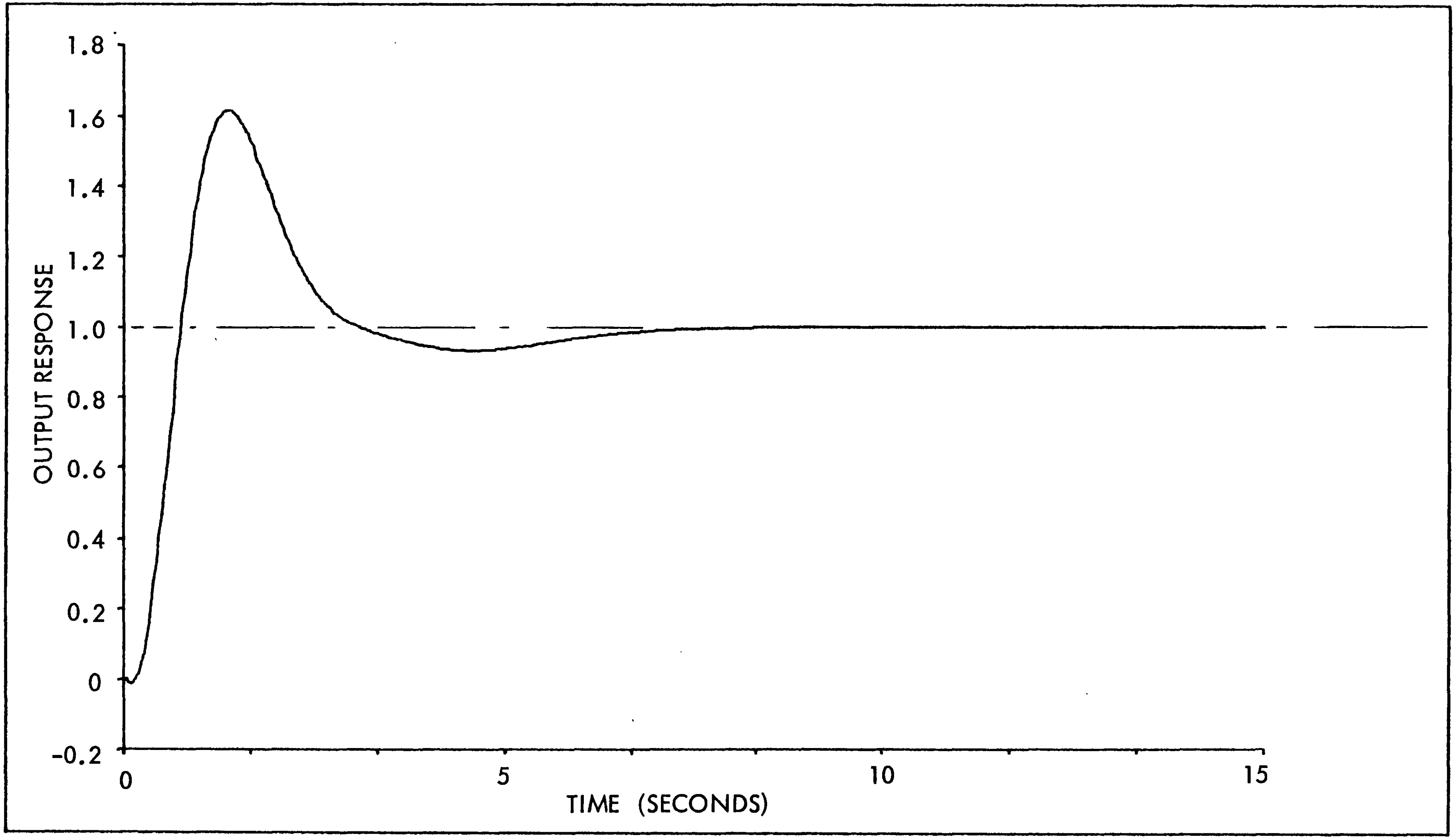


Fig.5.14. Unit Step Response for Cascade Realisation.

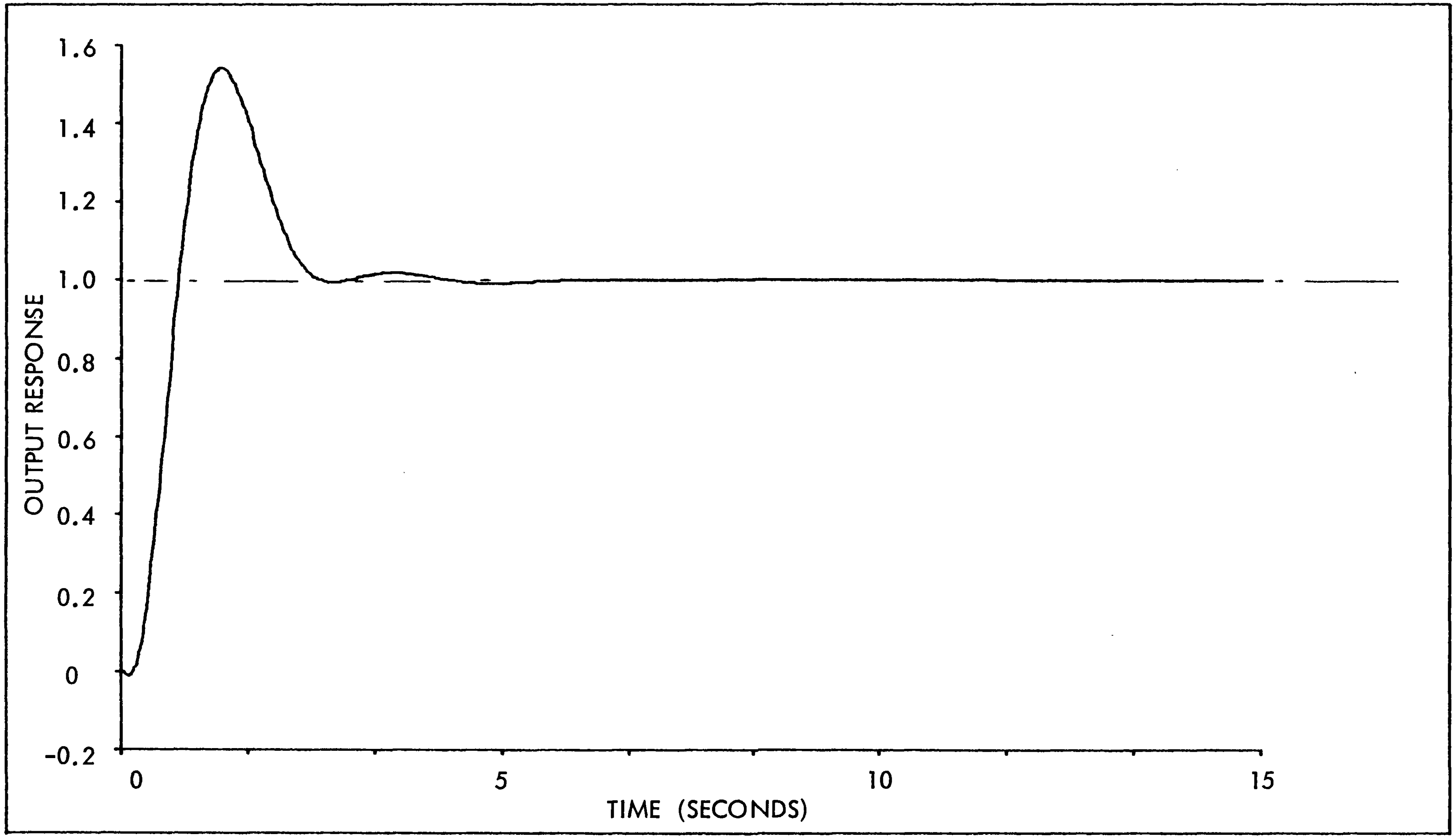


Fig.5.15. Unit Step Response for Cascade Realisation: Binary Rounded Coefficients.

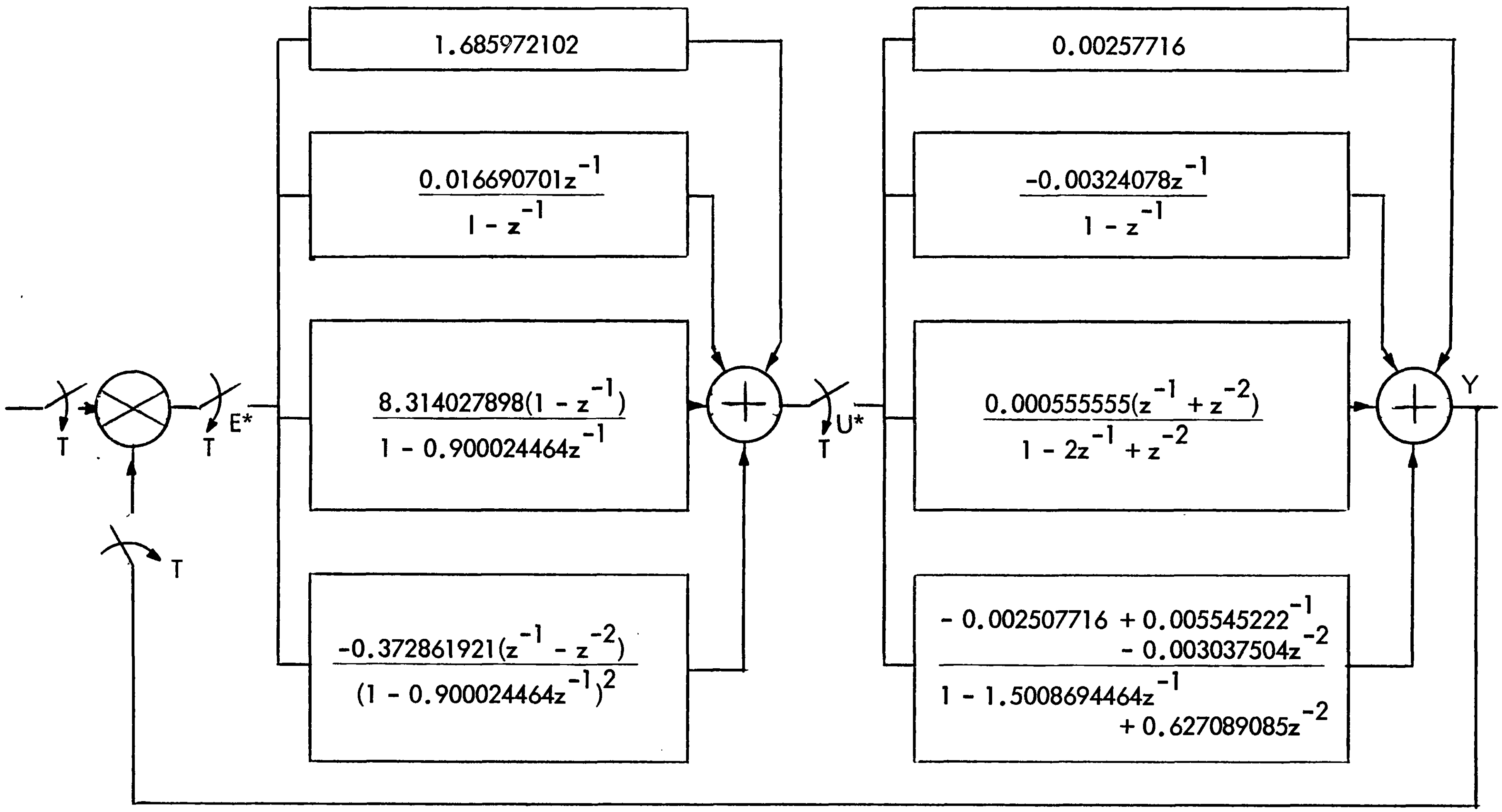

T
T
E*
1.685972102
$\frac{0.016690701z^{-1}}{1 - z^{-1}}$
$\frac{8.314027898(1 - z^{-1})}{1 - 0.900024464z^{-1}}$
$\frac{-0.372861921(z^{-1} - z^{-2})}{(1 - 0.900024464z^{-1})^2}$
T
U*
0.00257716
$\frac{-0.00324078z^{-1}}{1 - z^{-1}}$
$\frac{0.000555555(z^{-1} + z^{-2})}{1 - 2z^{-1} + z^{-2}}$
$\frac{- 0.002507716 + 0.005545222^{-1} - 0.003037504z^{-2}}{1 - 1.5008694464z^{-1} + 0.627089085z^{-2}}$
Y
T

Fig.5.16. Parallel Realisation of Guidance Loop.

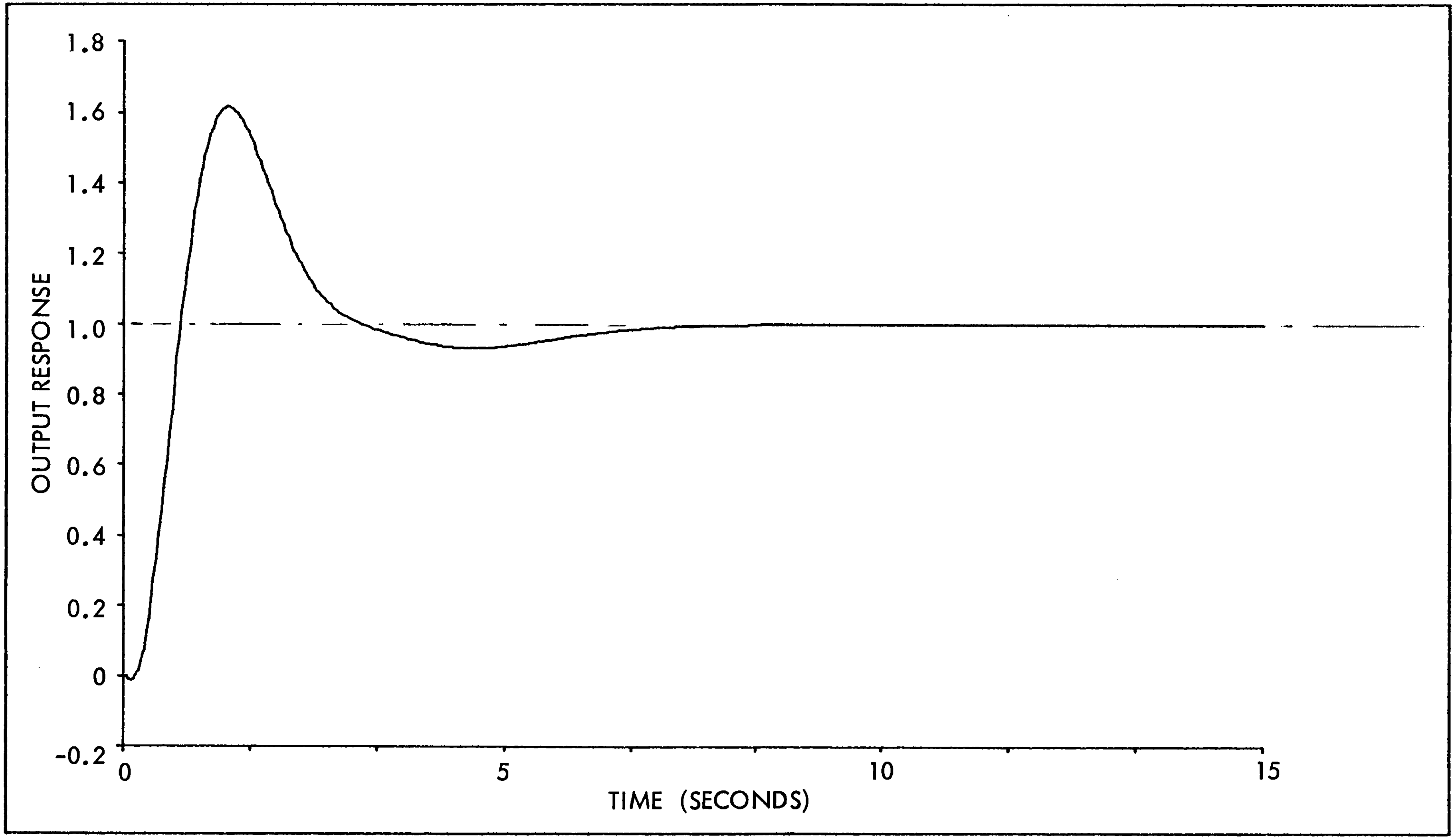


Fig.5.17. Unit Step Response for Parallel Realisation.

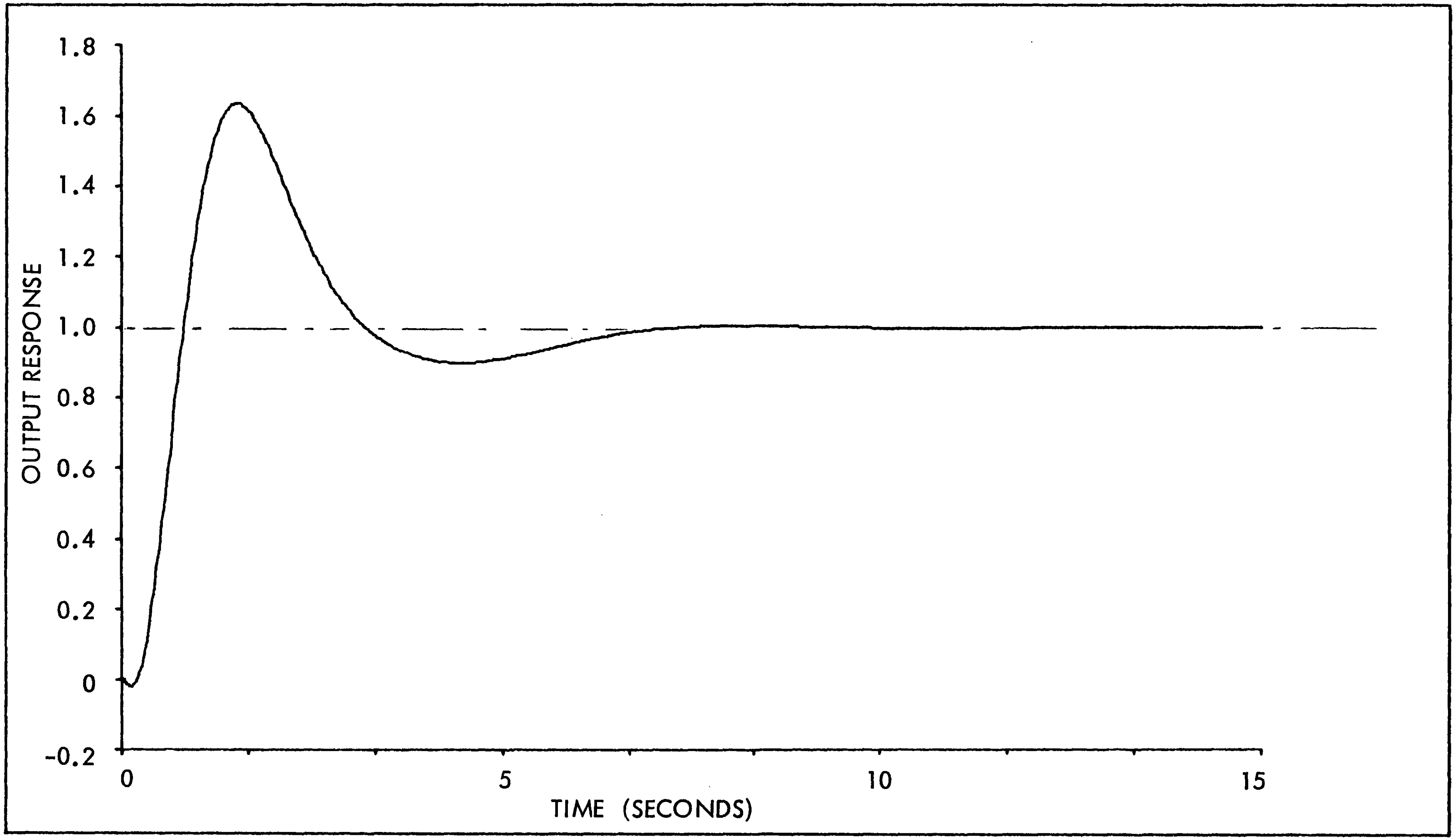


Fig.5.18. Unit Step Response for Parallel Realisation: Binary Rounded Coefficients.

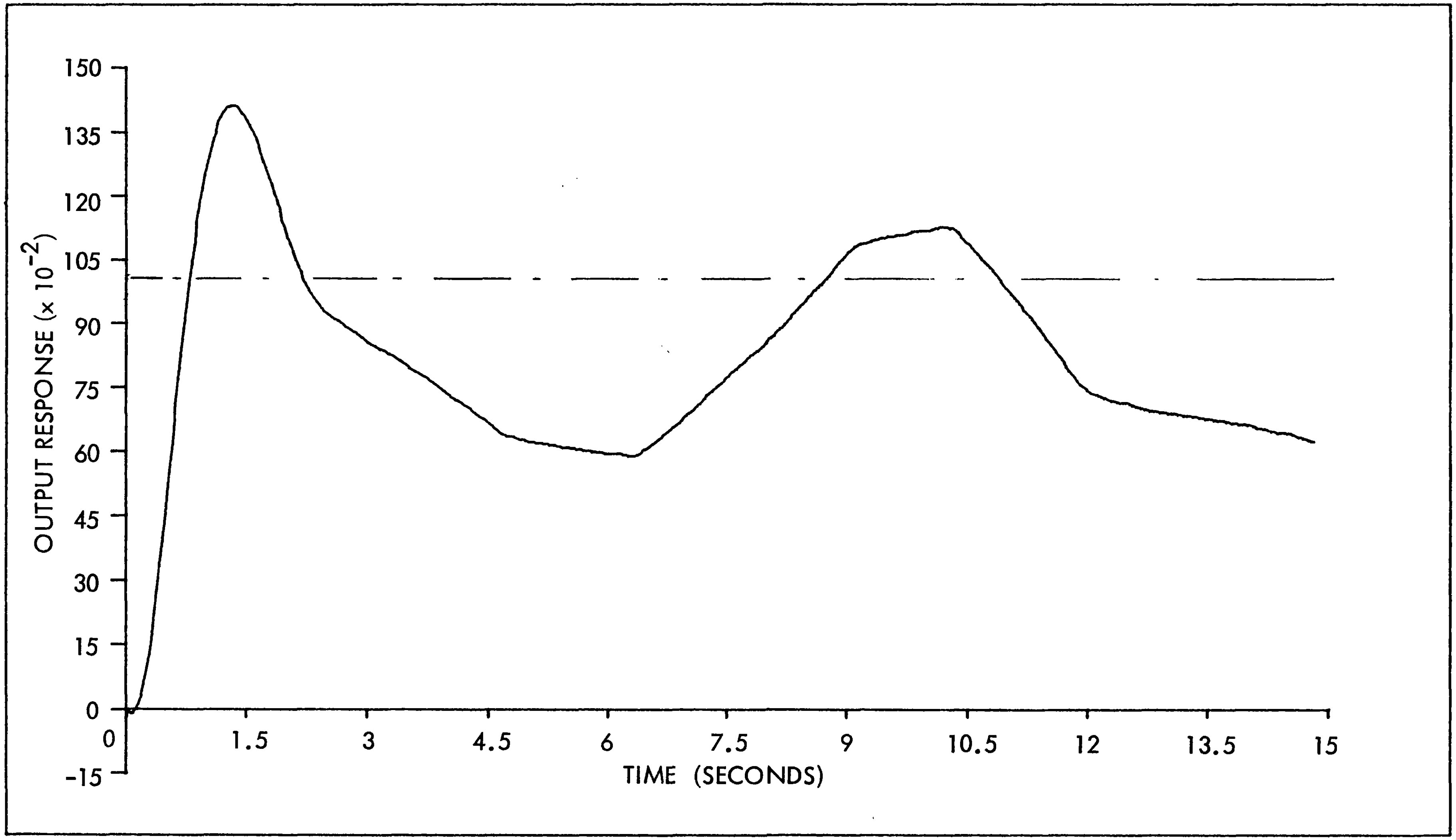


Fig.5.19. Unit Step Response for Parallel Realisation: 16 Bit Integer Arithmetic.

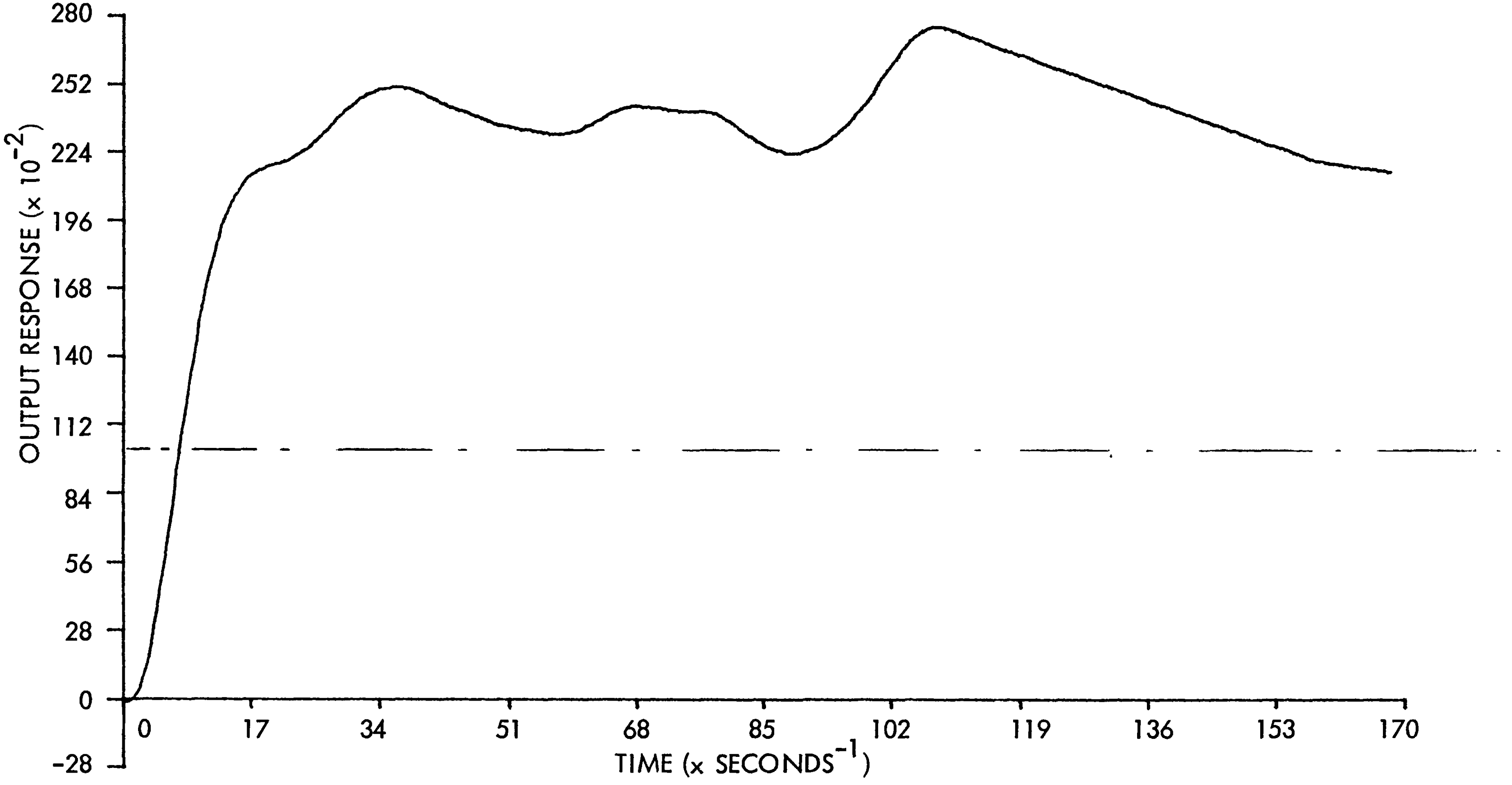


Fig.5.20. Unit Step Response for Cascade Realisation: 16 Bit Integer Arithmetic.

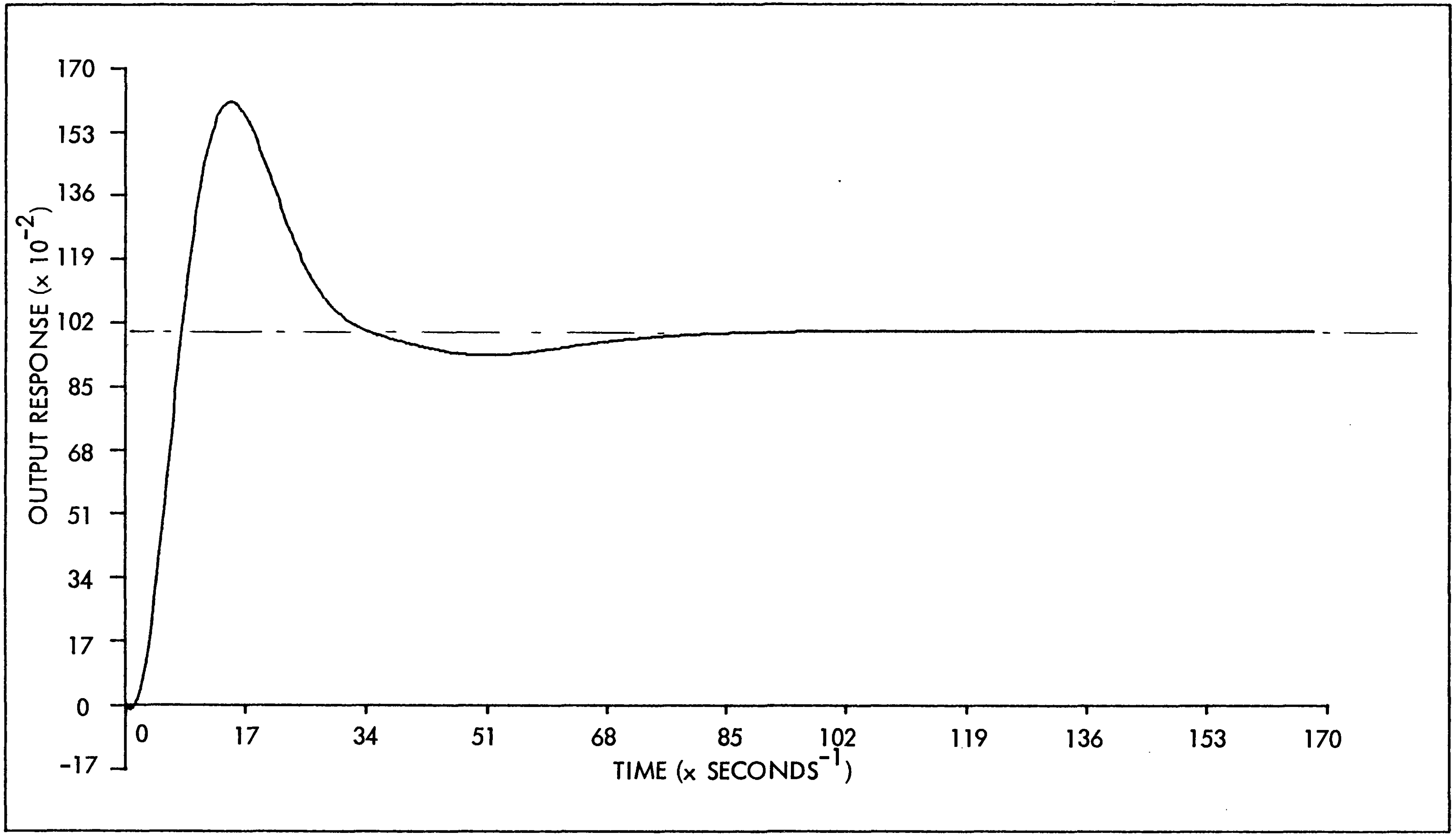


Fig.5.21. Unit Step Response for Parallel Realisation: 32 Bit Integer Arithmetic.

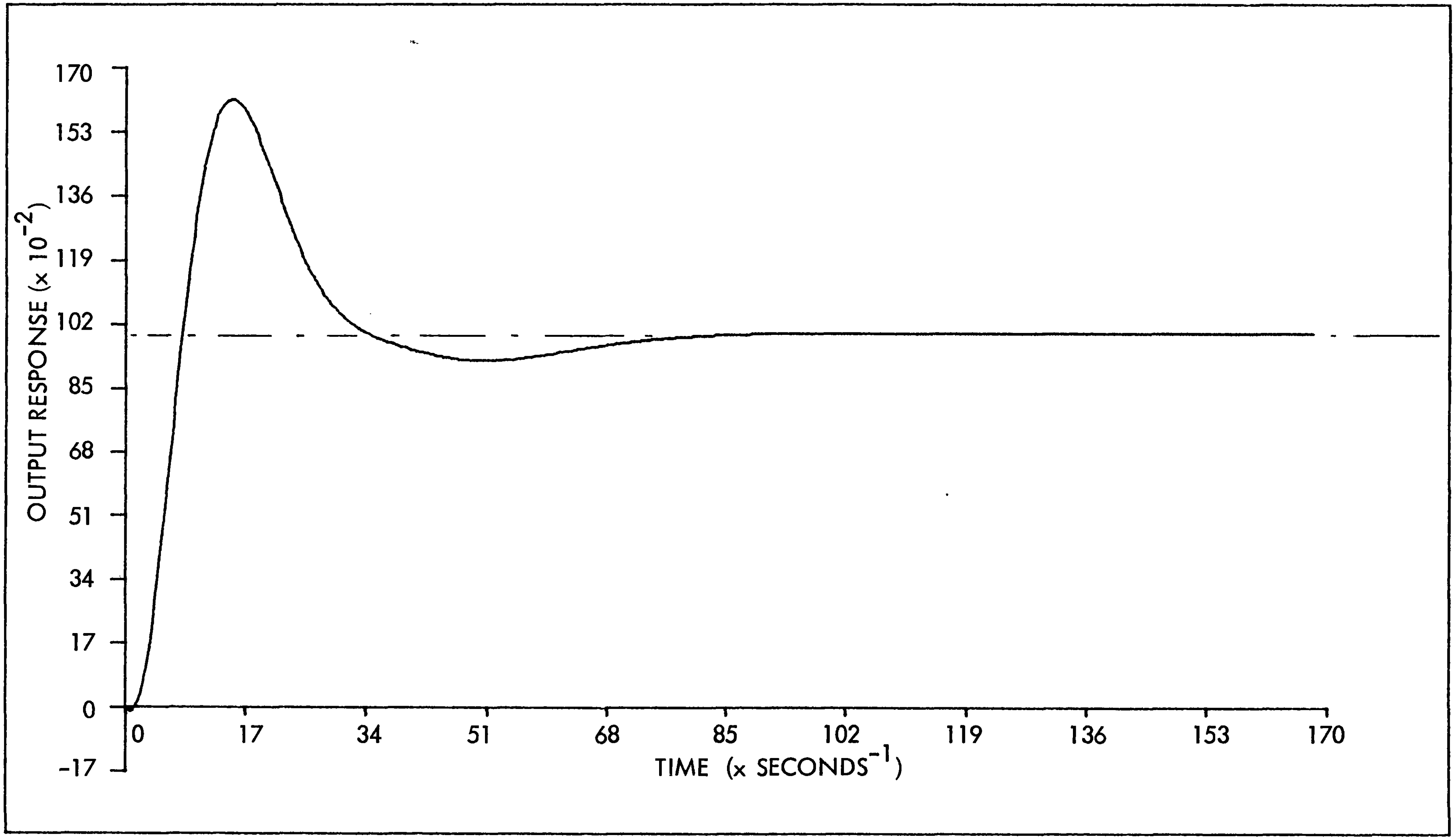


Fig.5.22. Unit Step Response for Cascade Realisation: 32 Bit Integer Arithmetic.

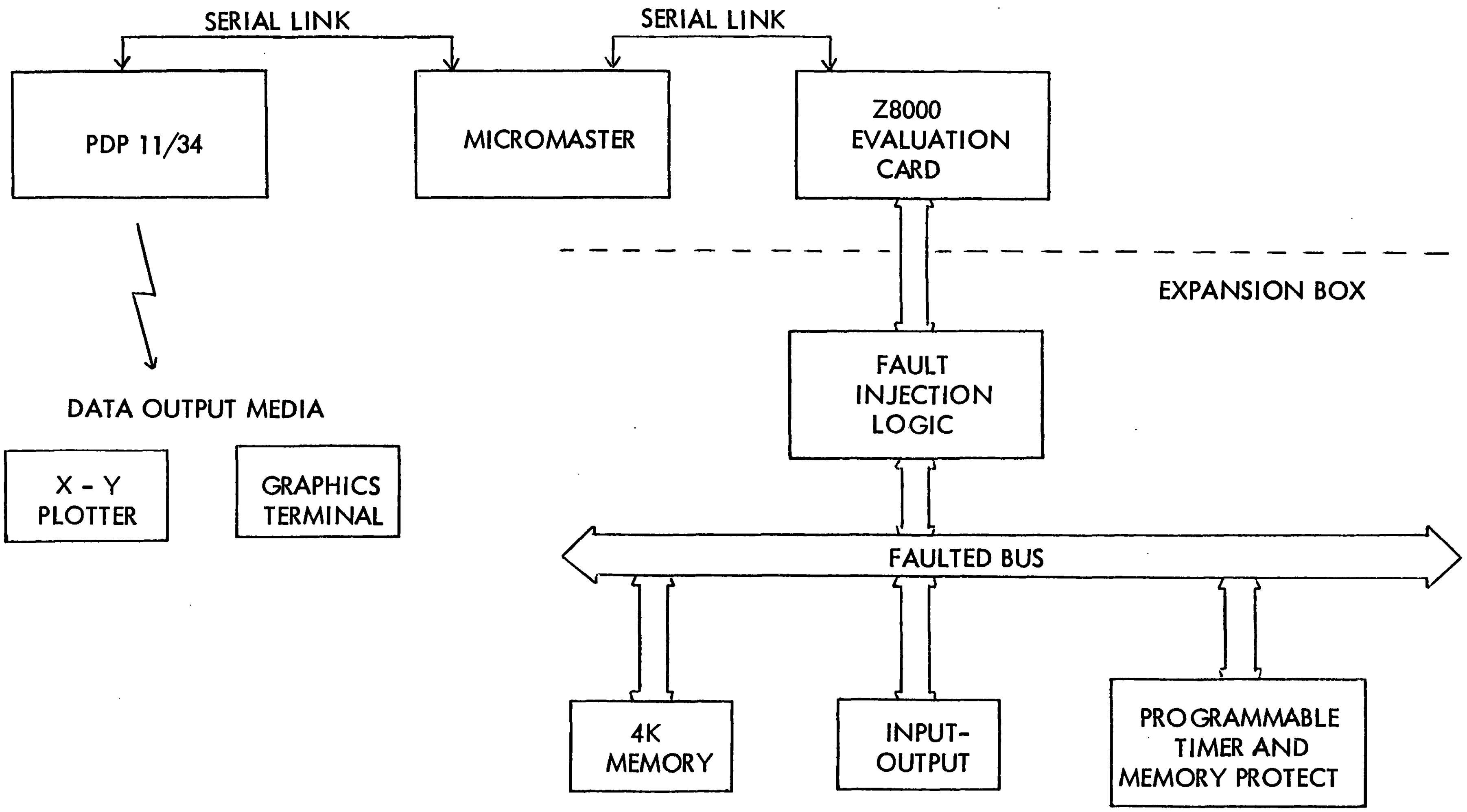


Fig.6.1. Overall Systems Diagram.

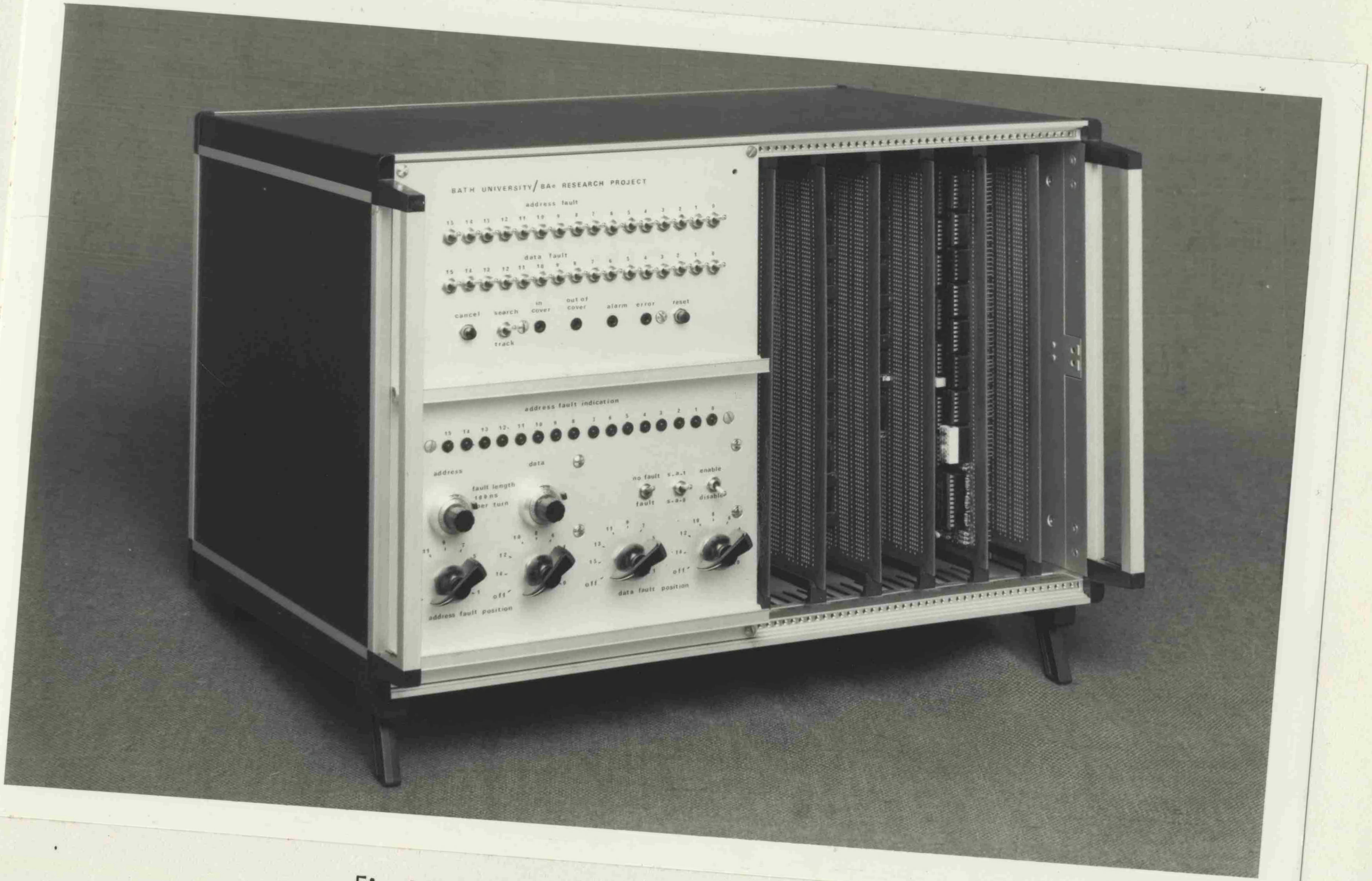
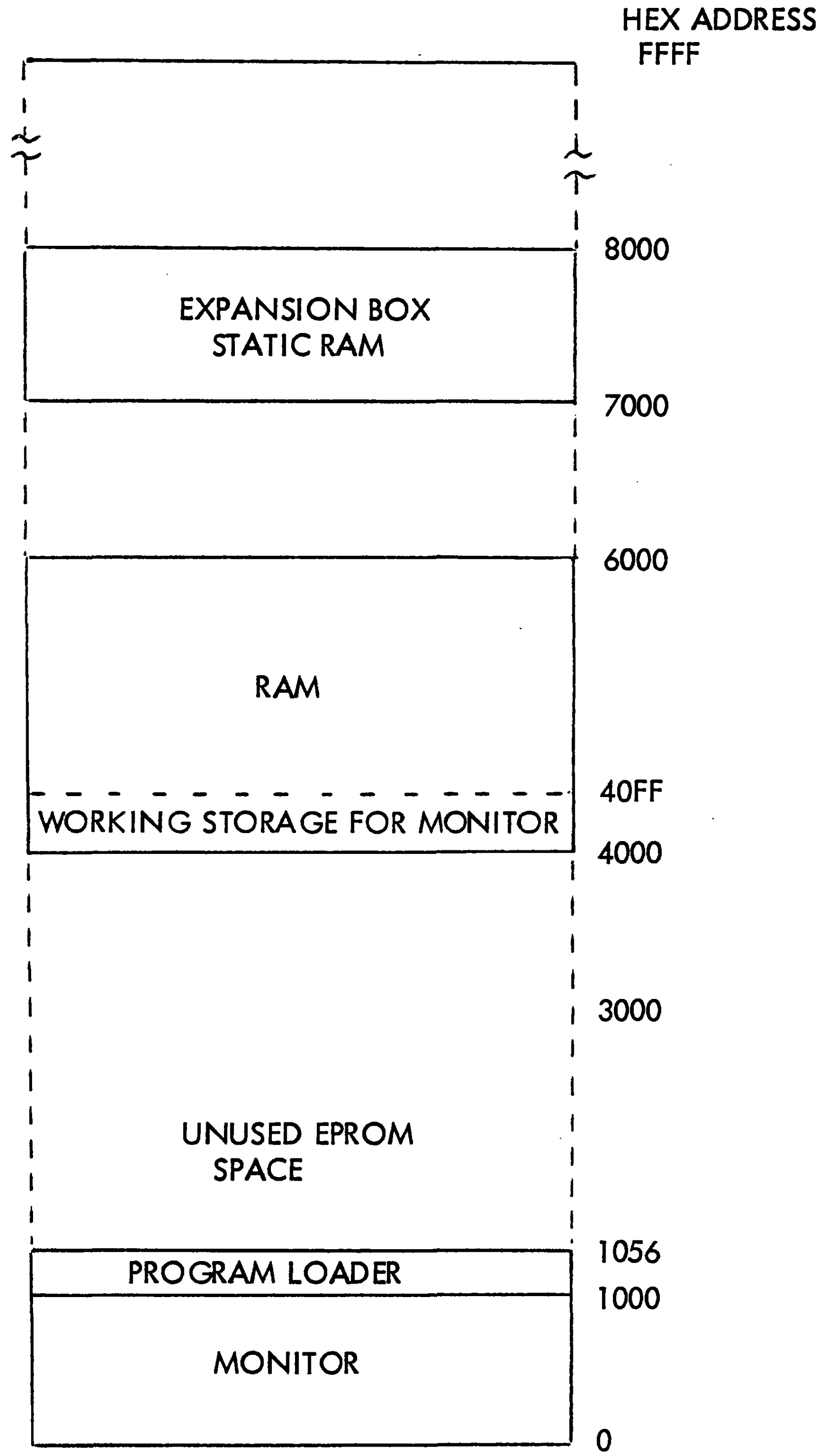
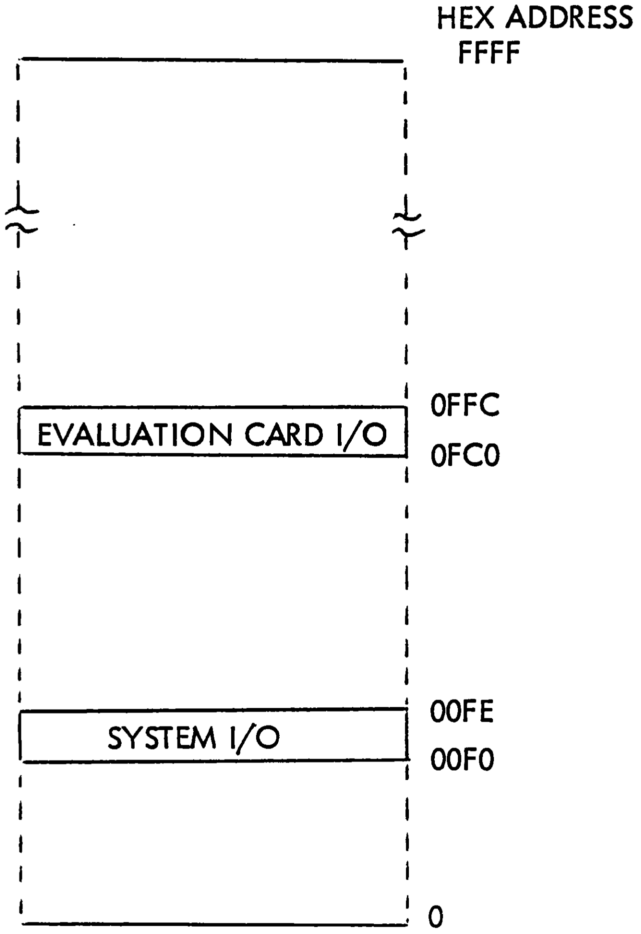


Fig.6.2. Microprocessor Expansion Box.



EVALUATION BOARD MEMORY : 0 - 6000 (HEX)

Fig.6.3. Memory Map.



SYSTEM I/O

00FE	-	TIMER
00FC	-	SEARCH/TRACK SWITCH
00FA	-	CANCEL PUSH BUTTON
00F8	-	ALARM LED
00F6	-	IN COVER LED
00F4	-	OUT OF COVER LED
00F2	-	ERROR LED
00F0	-	TIME OUT RESET

Fig.6.4. Input/Output Map.

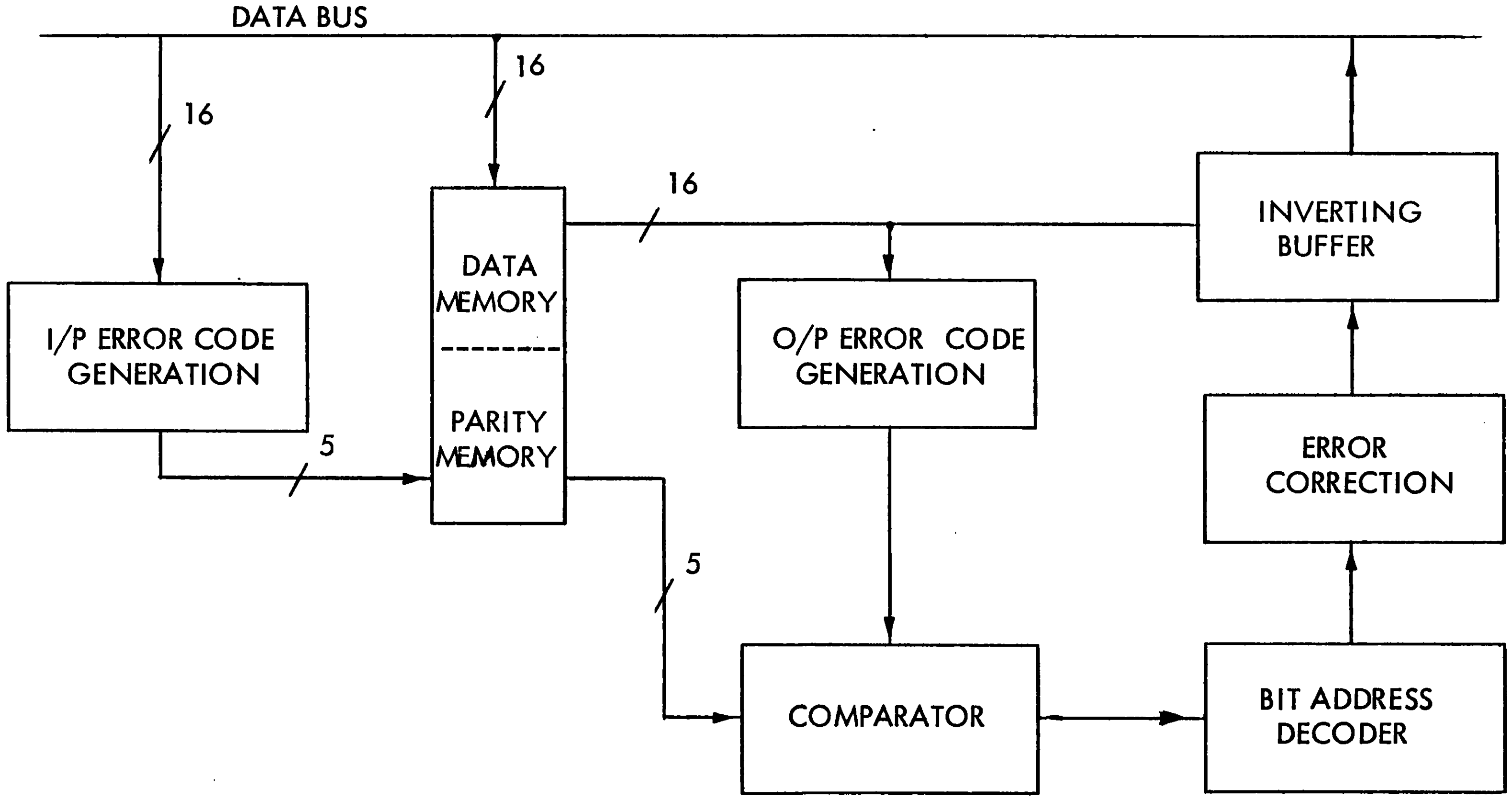


Fig.6.5. Schematic of Error Correcting Memory.

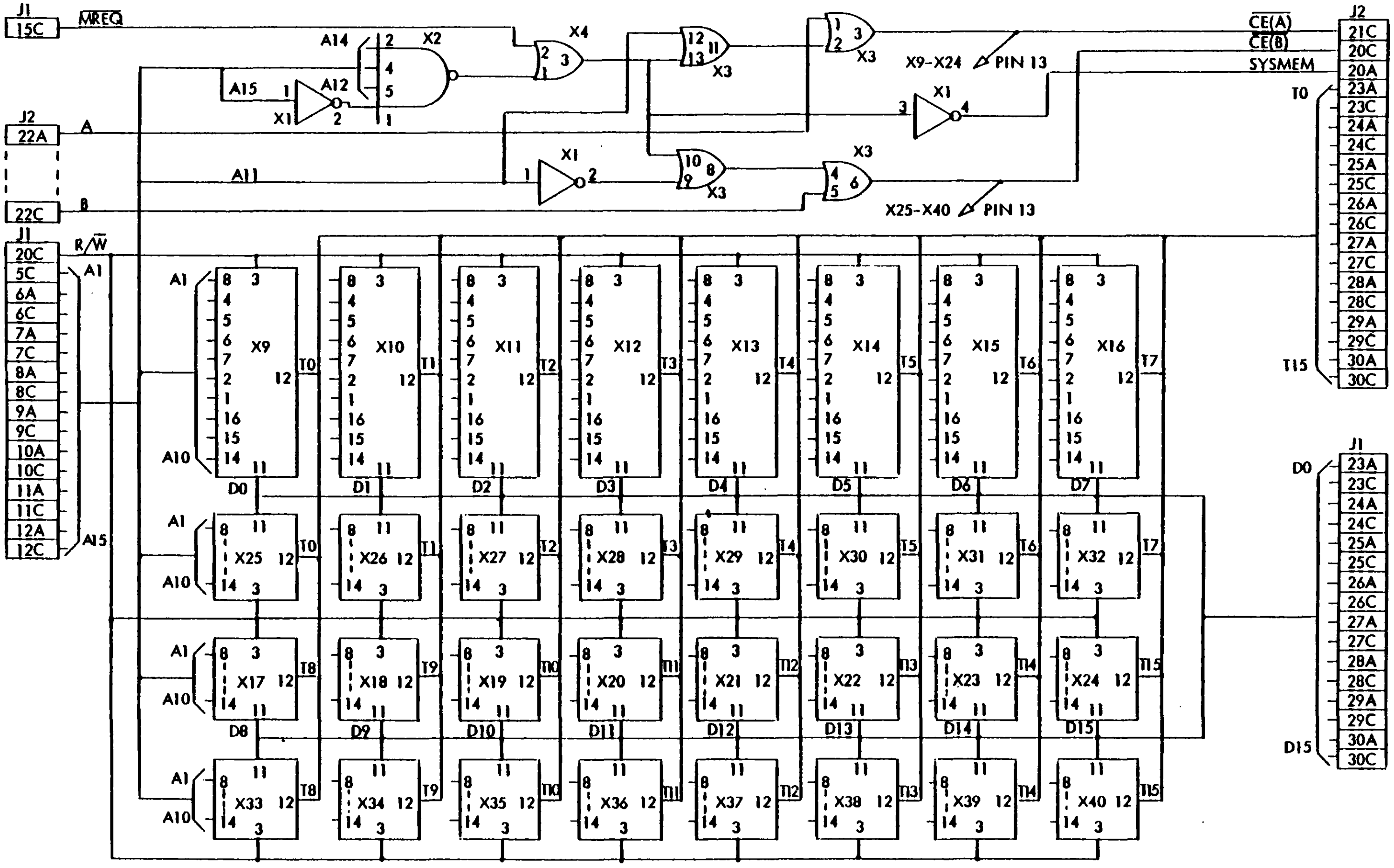


Fig. 6.6. Error Correcting Memory Board 1.

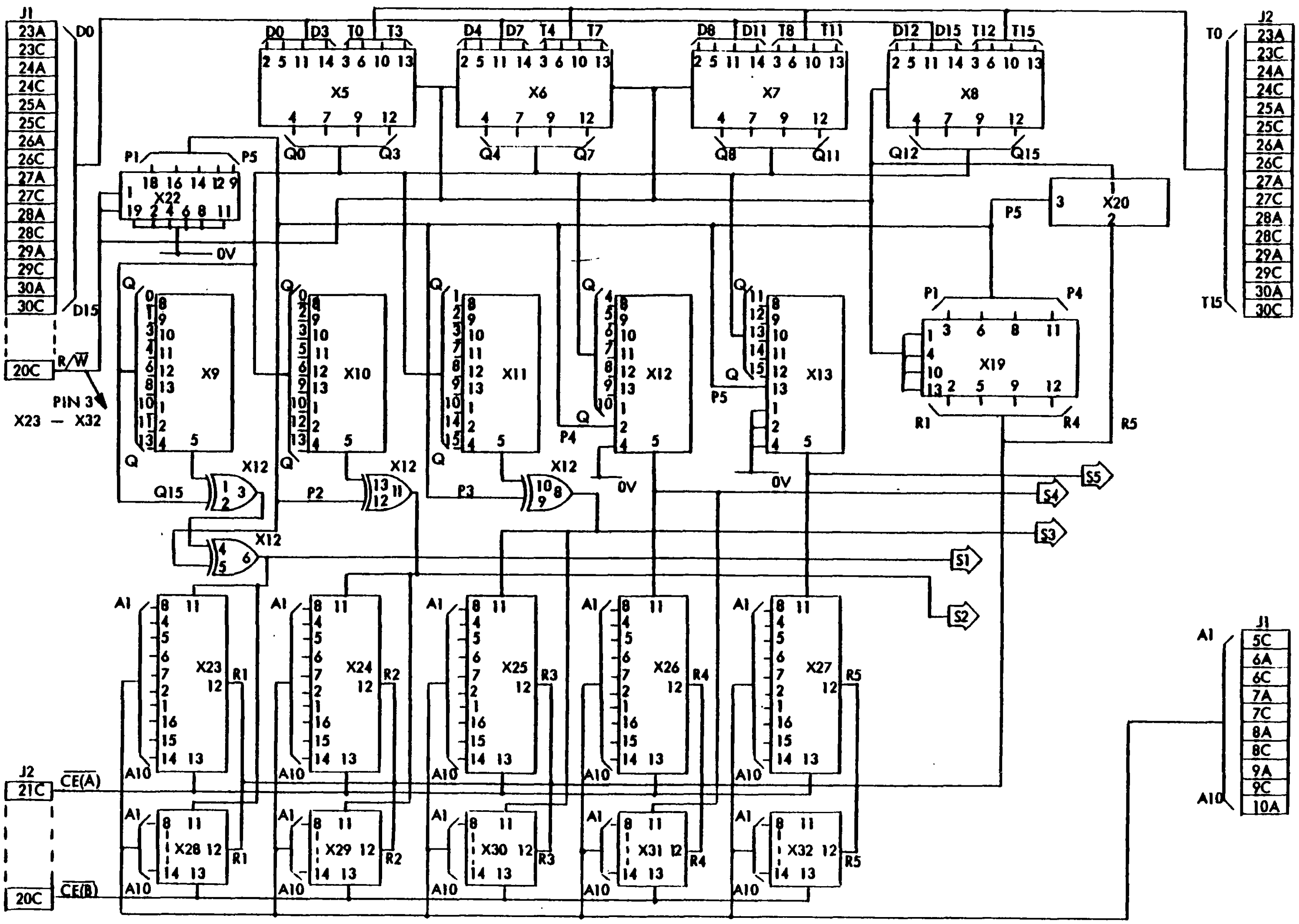


Fig.6.7.a. Error Correcting Memory Board 2.

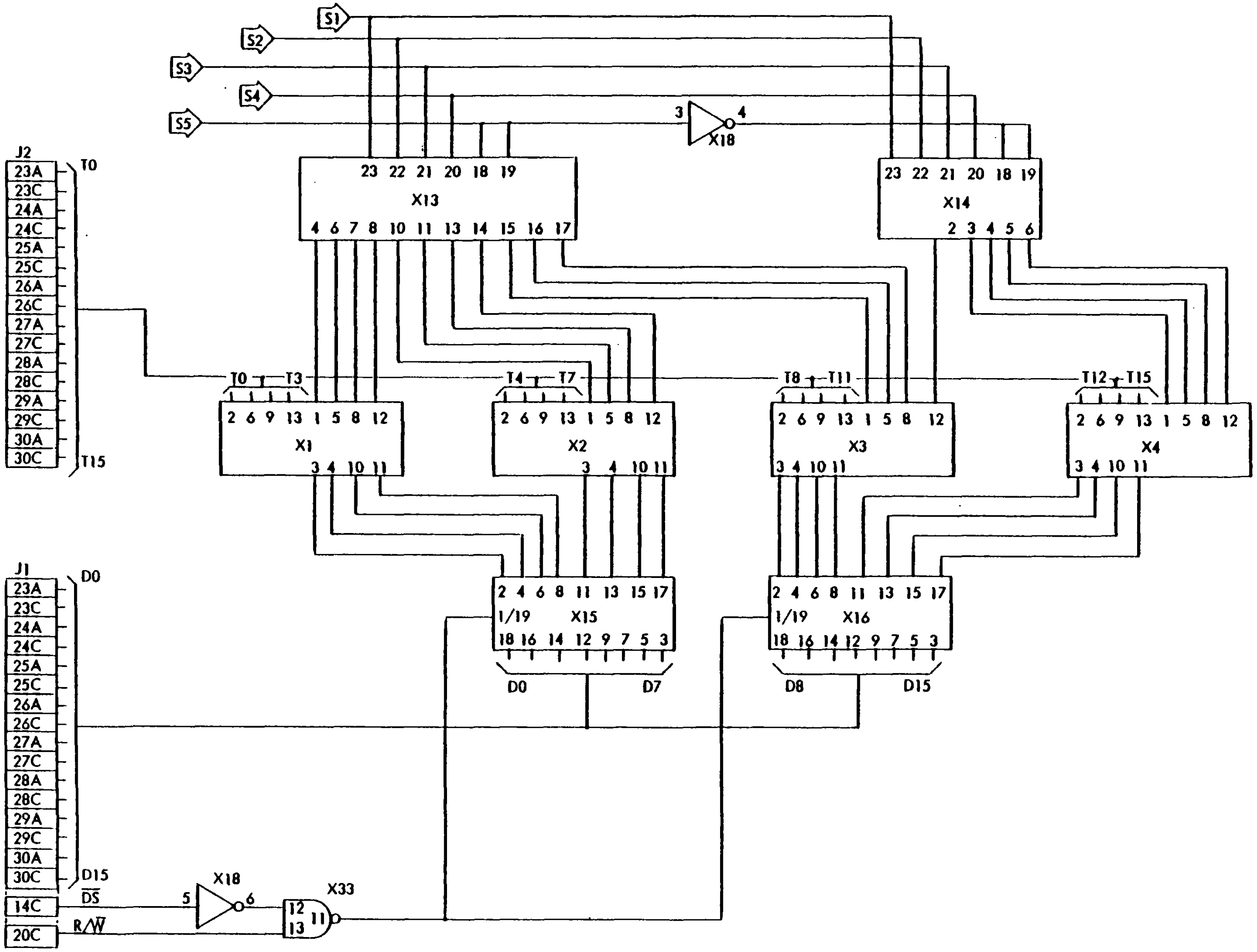


Fig.6.7.b. Error Correcting Memory Board 2.

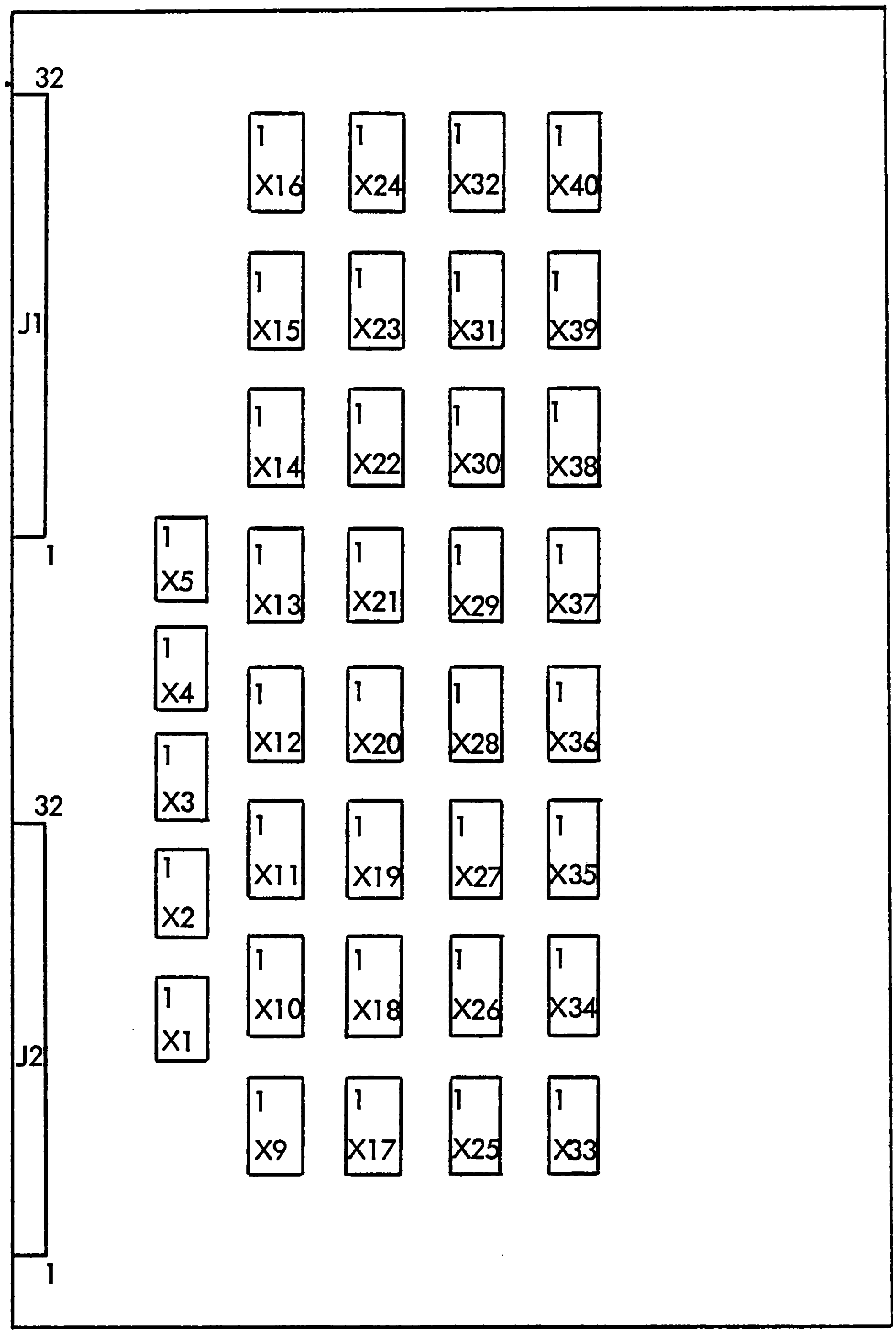


Fig.6.8. Layout of Error Correcting Memory Board 1.

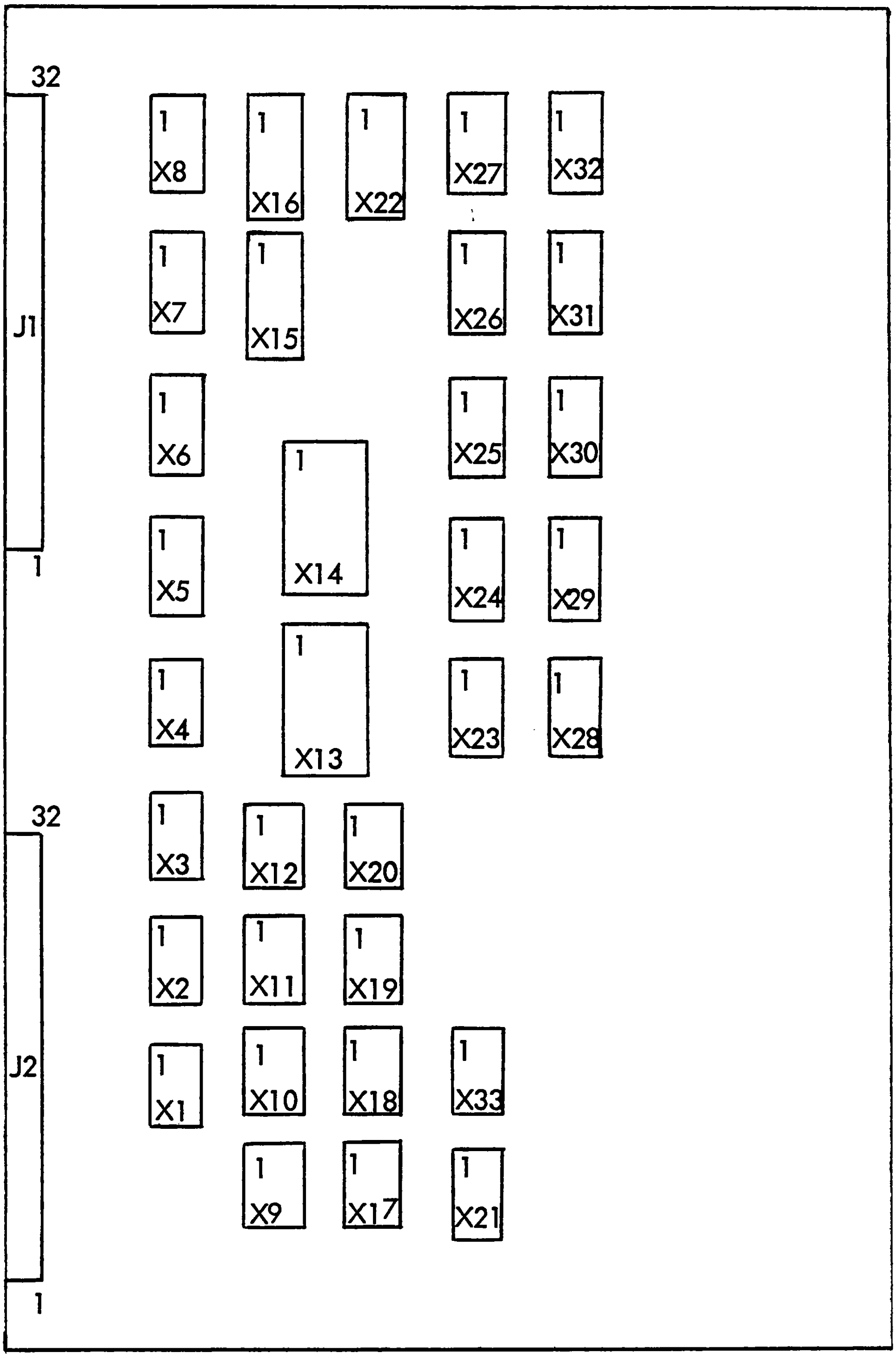


Fig.6.9. Layout of Error Correcting Memory Board 2.

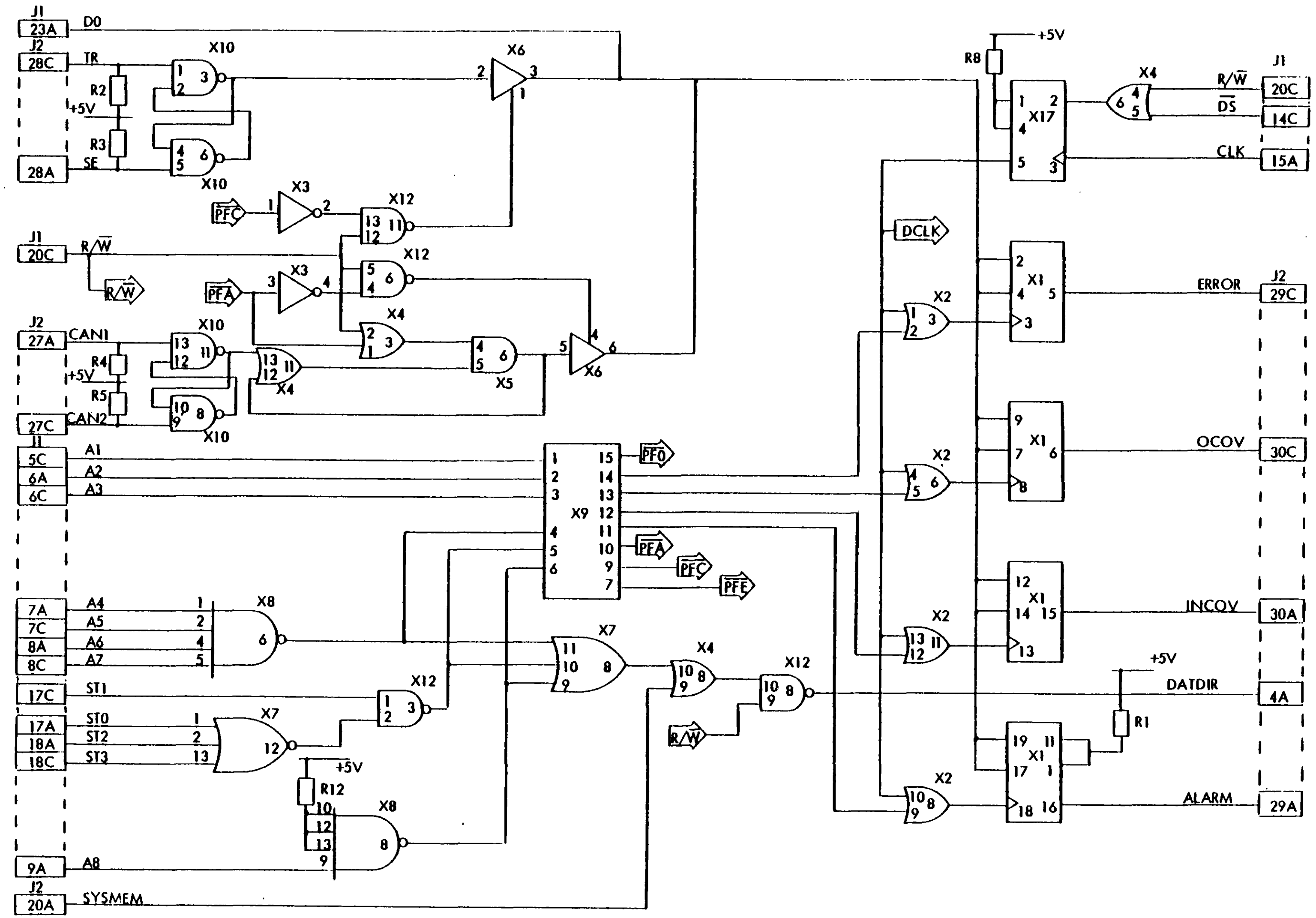


Fig. 6. 10. a. Input/Output Board.

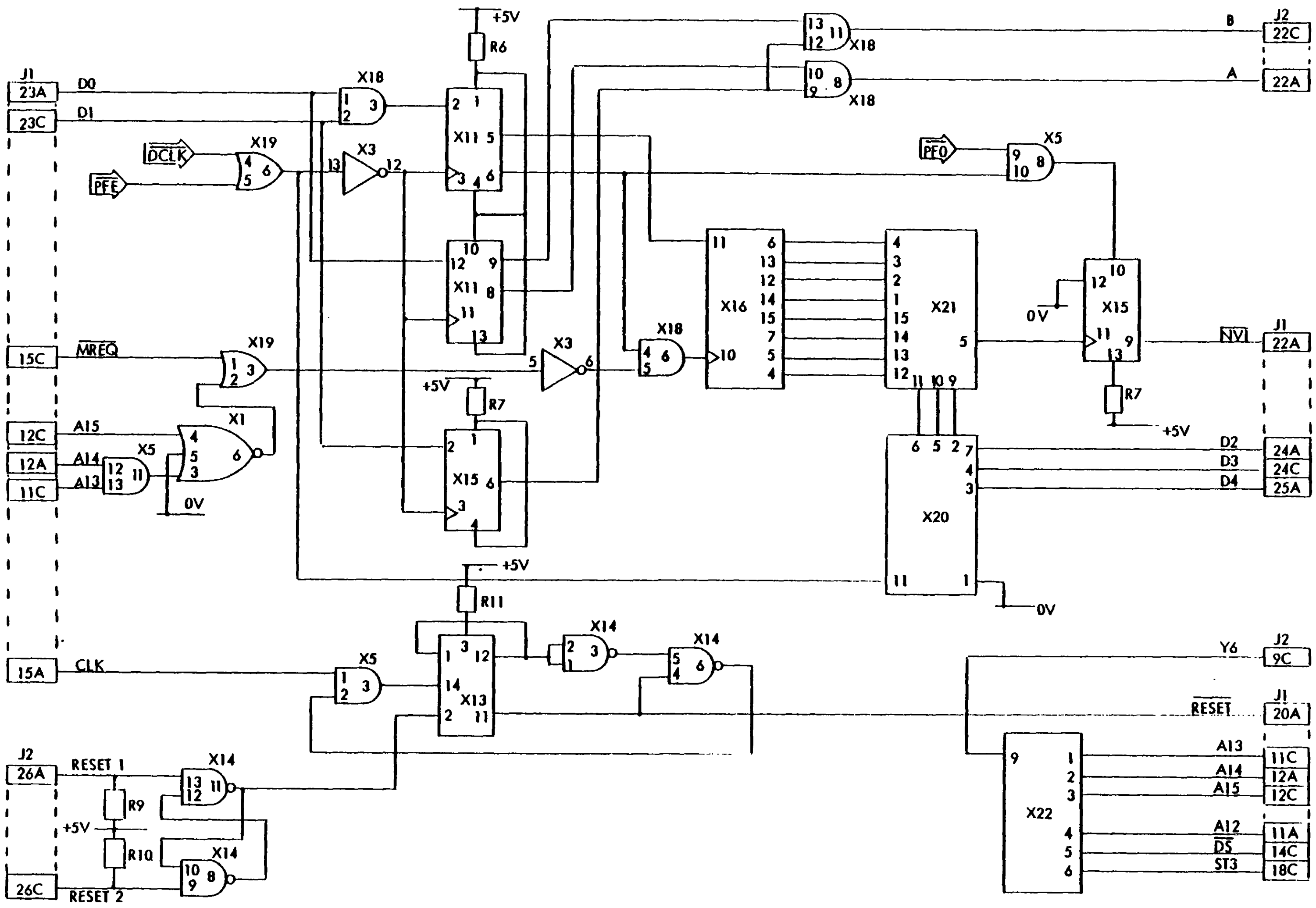


Fig.6.10.b. Input/Output Board.

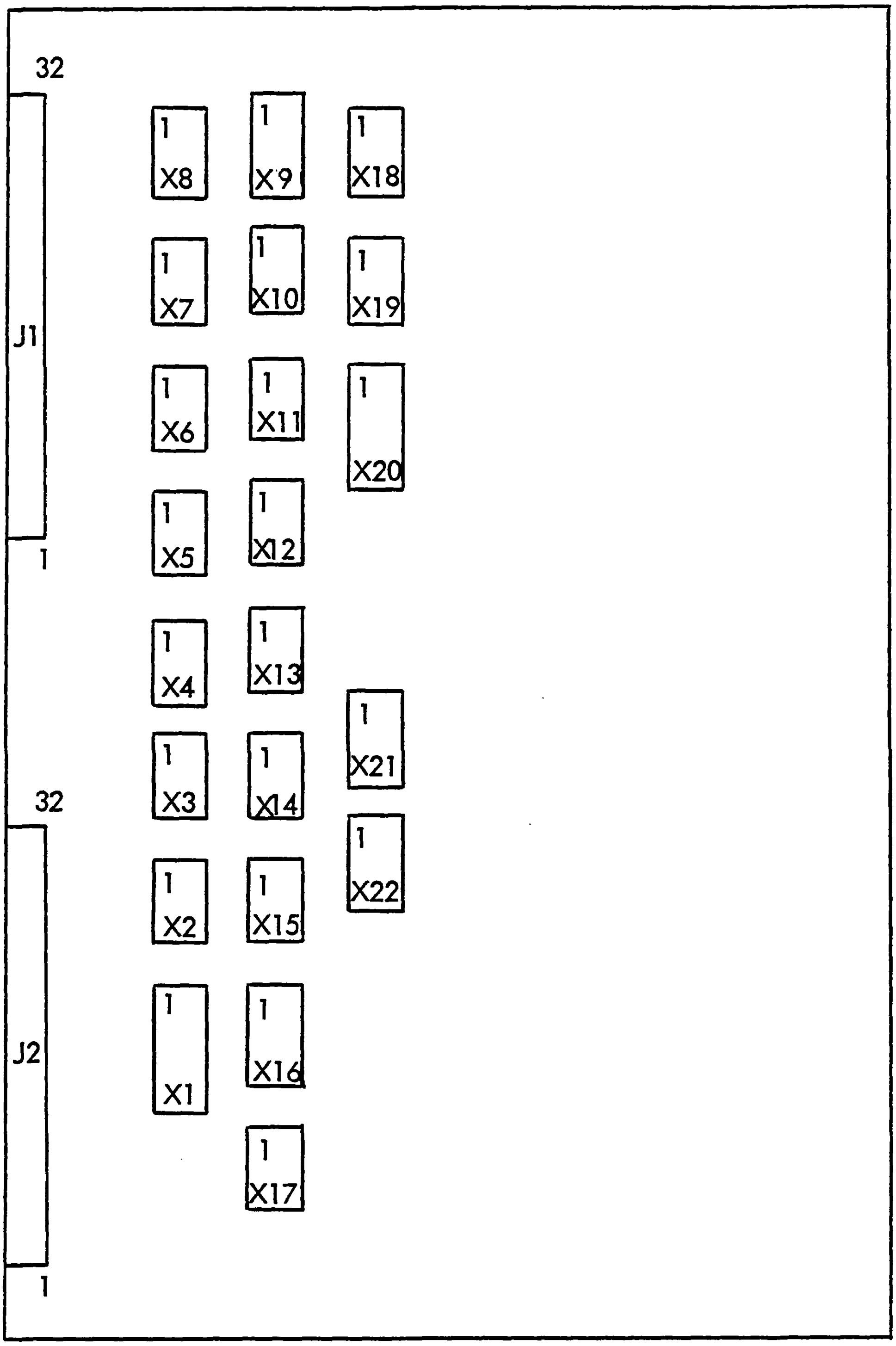


Fig.6.11. Layout of Input/Output Board.

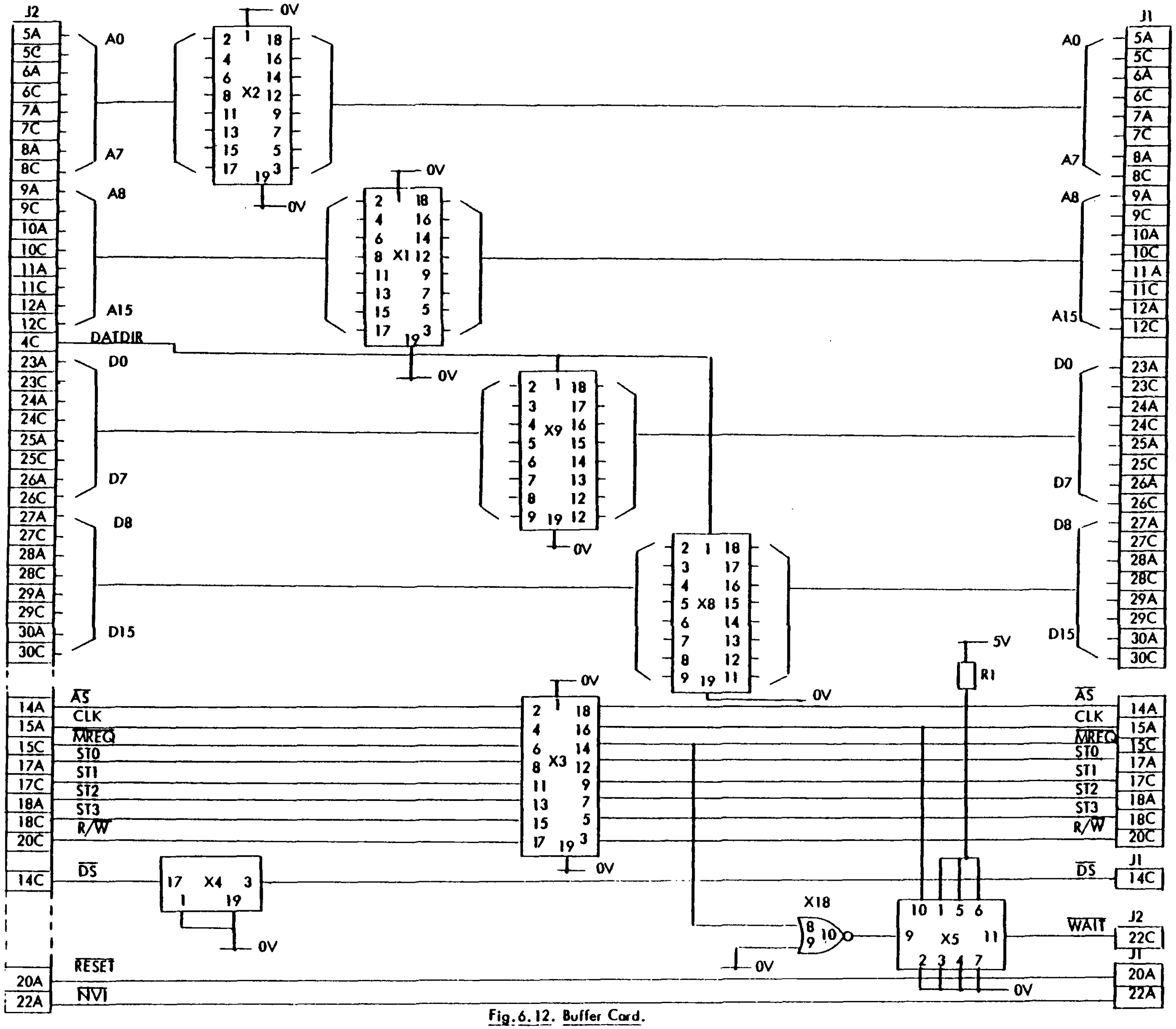


Fig. 6. 12. Buffer Card.

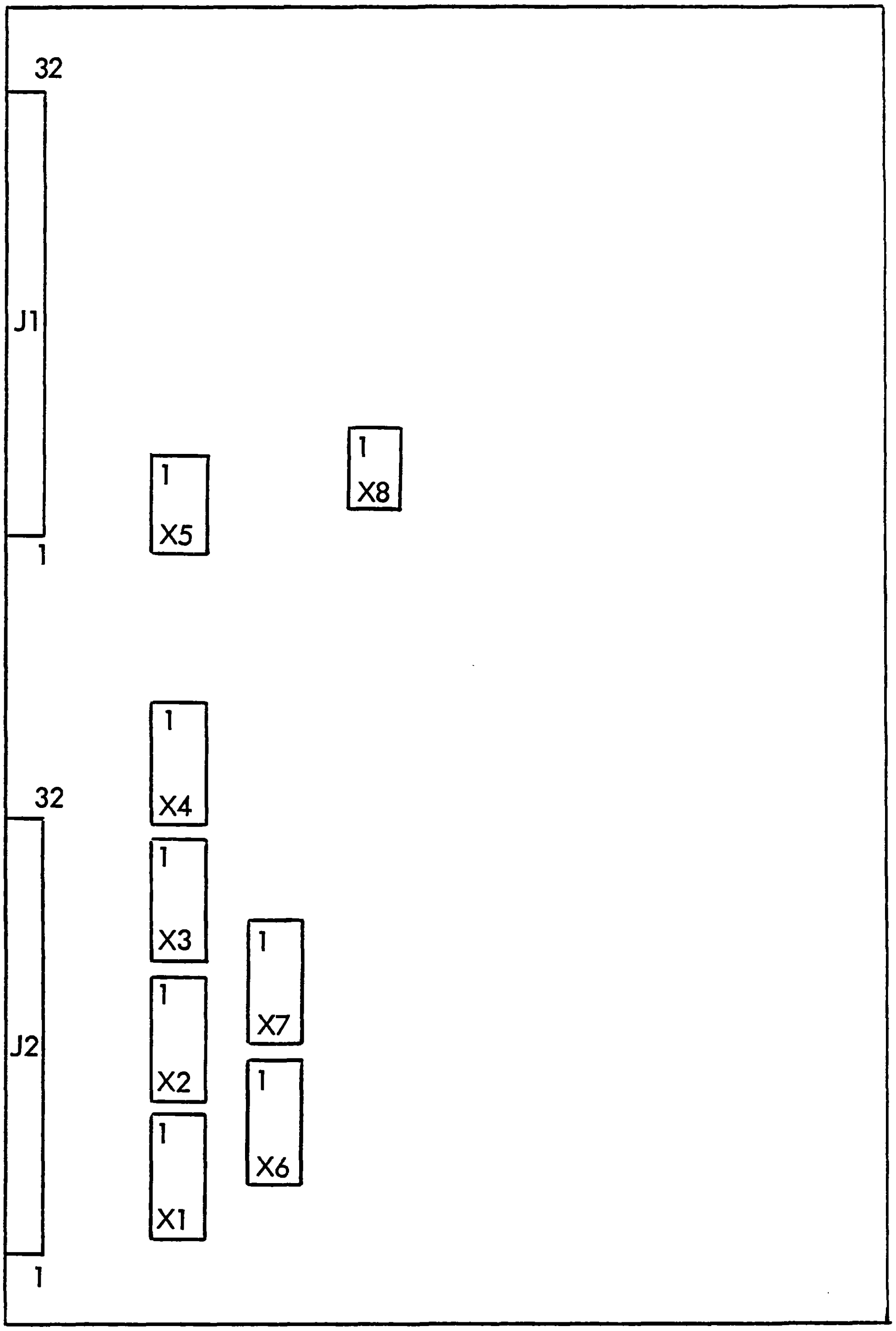


Fig.6.13. Layout of Buffer Card.

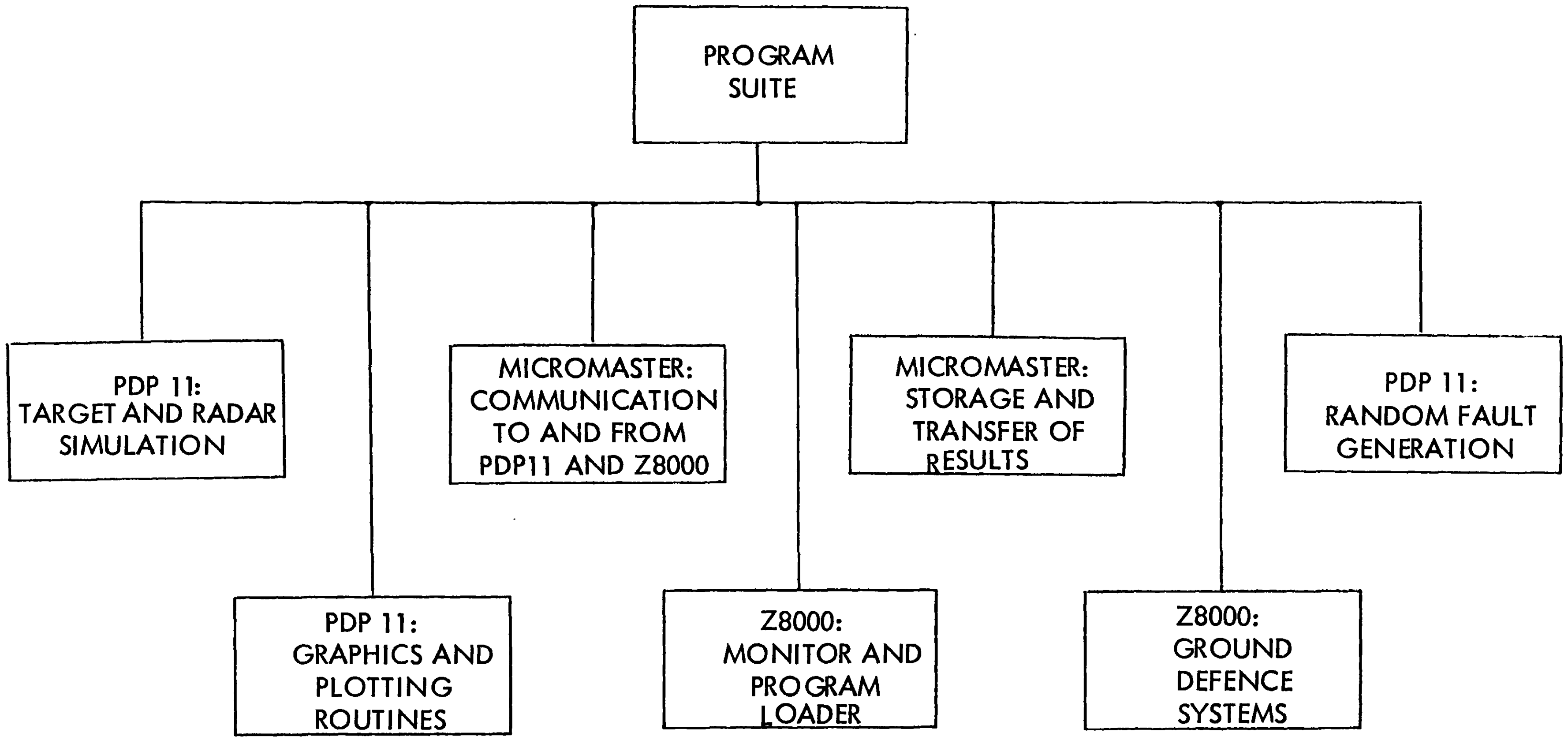

PROGRAM SUITE
PDP 11: TARGET AND RADAR SIMULATION
PDP 11: GRAPHICS AND PLOTTING ROUTINES
MICROMASTER: COMMUNICATION TO AND FROM PDP11 AND Z8000
Z8000: MONITOR AND PROGRAM LOADER
MICROMASTER: STORAGE AND TRANSFER OF RESULTS
Z8000: GROUND DEFENCE SYSTEMS
PDP 11: RANDOM FAULT GENERATION

Fig.6.14. System Software Suite.

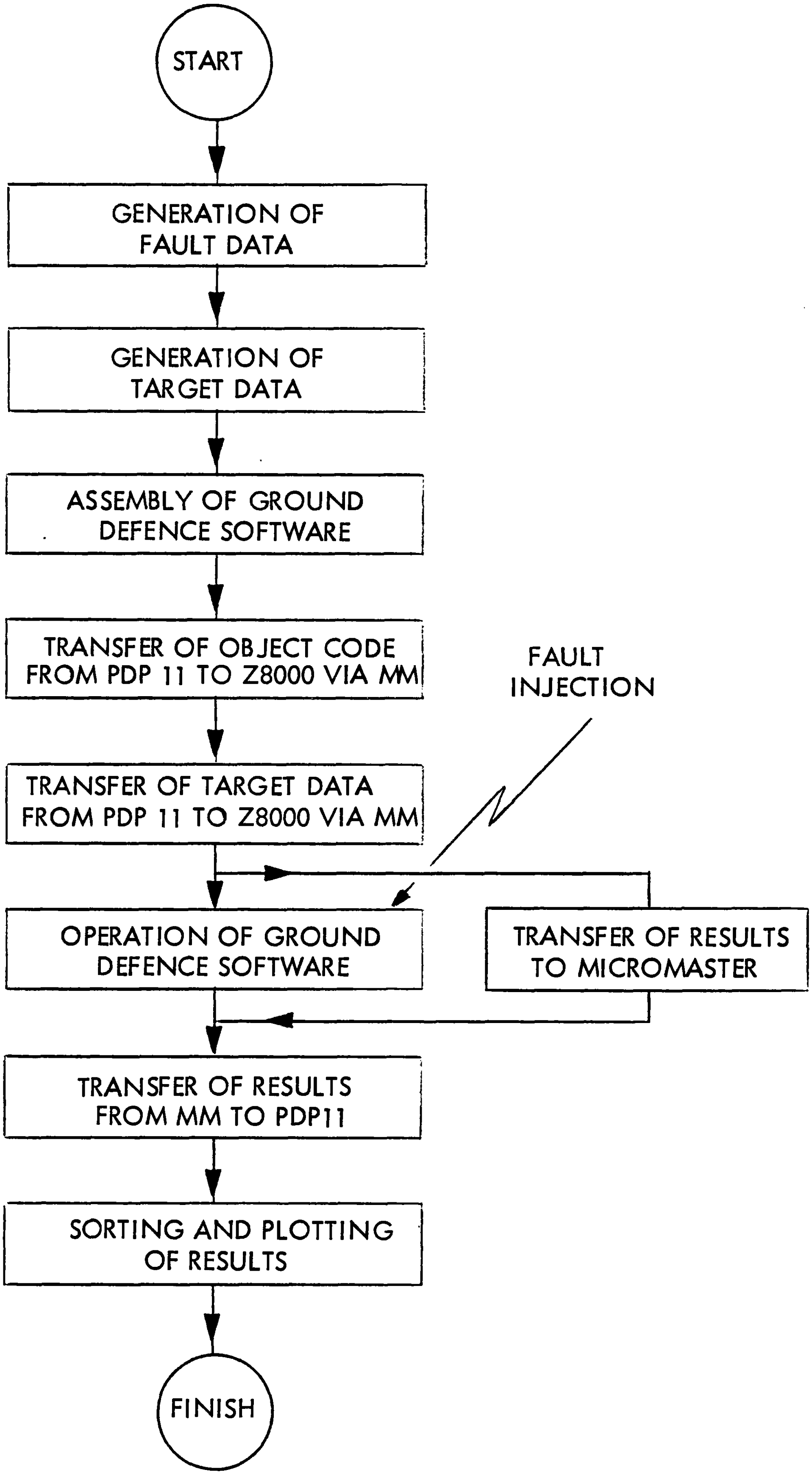

START
GENERATION OF FAULT DATA
GENERATION OF TARGET DATA
ASSEMBLY OF GROUND DEFENCE SOFTWARE
TRANSFER OF OBJECT CODE FROM PDP 11 TO Z8000 VIA MM
TRANSFER OF TARGET DATA FROM PDP 11 TO Z8000 VIA MM
FAULT INJECTION
OPERATION OF GROUND DEFENCE SOFTWARE
TRANSFER OF RESULTS TO MICROMASTER
TRANSFER OF RESULTS FROM MM TO PDP11
SORTING AND PLOTTING OF RESULTS
FINISH

Fig.6.15. System Software Typical Operation.

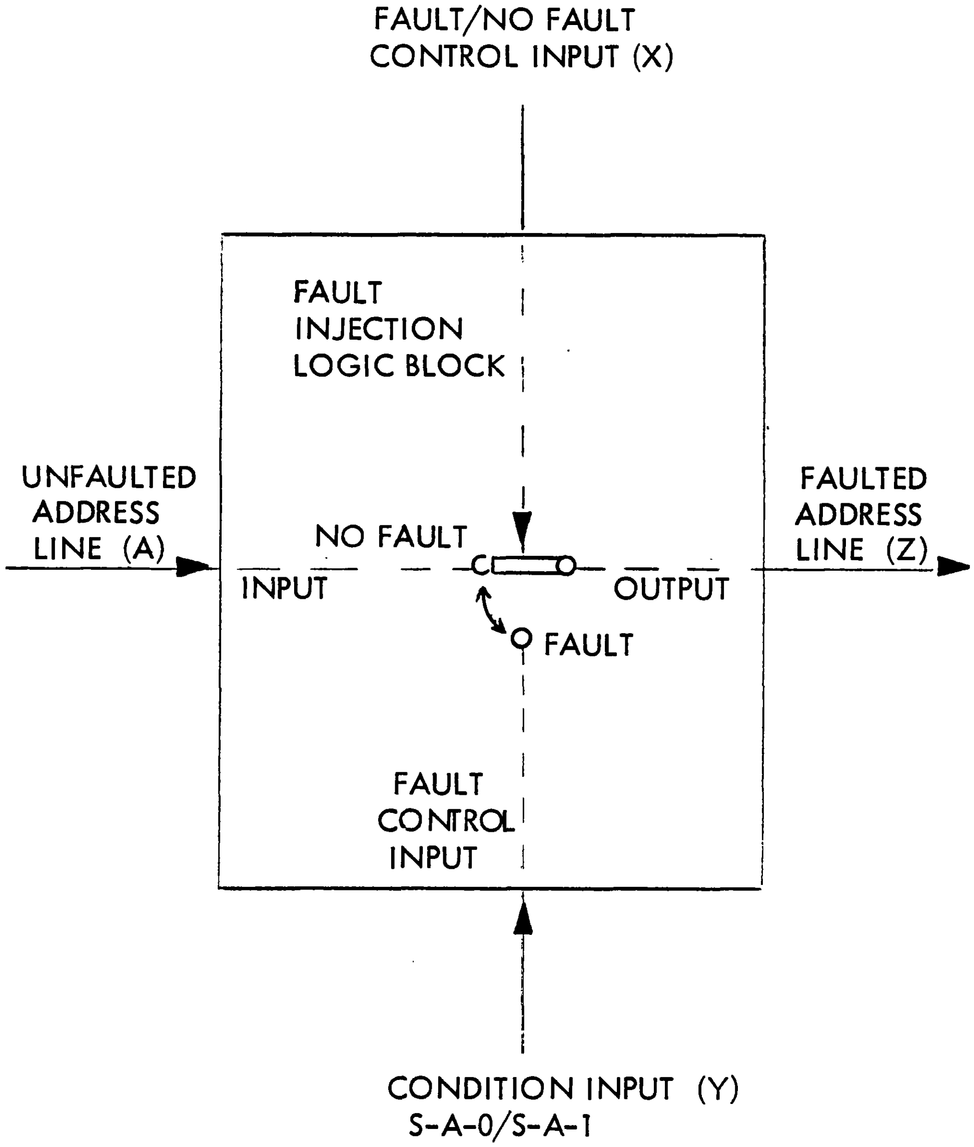


Fig.6.16. Schematic of Fault Injection Logic.

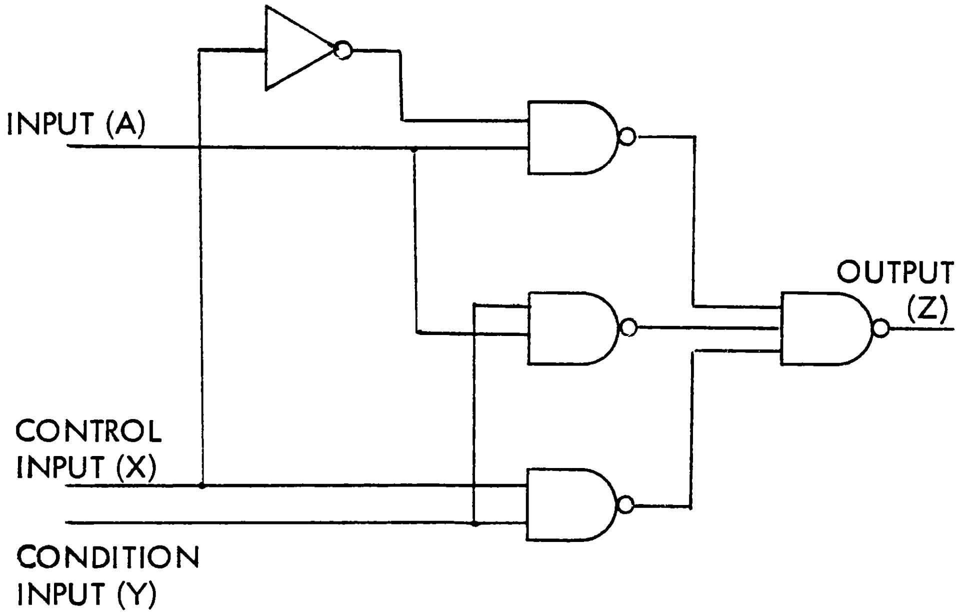


Fig.6.17. Implementation of Address Fault Logic.

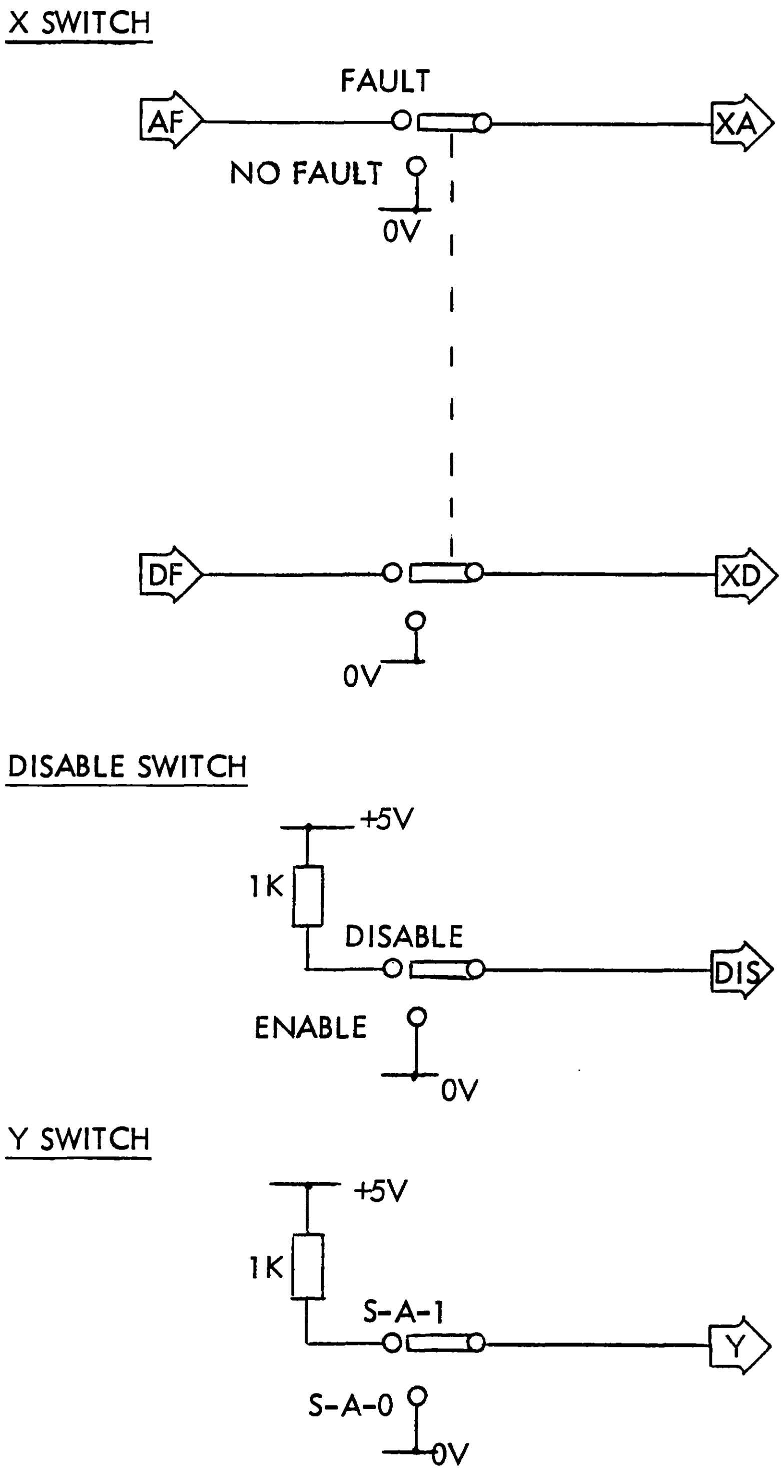
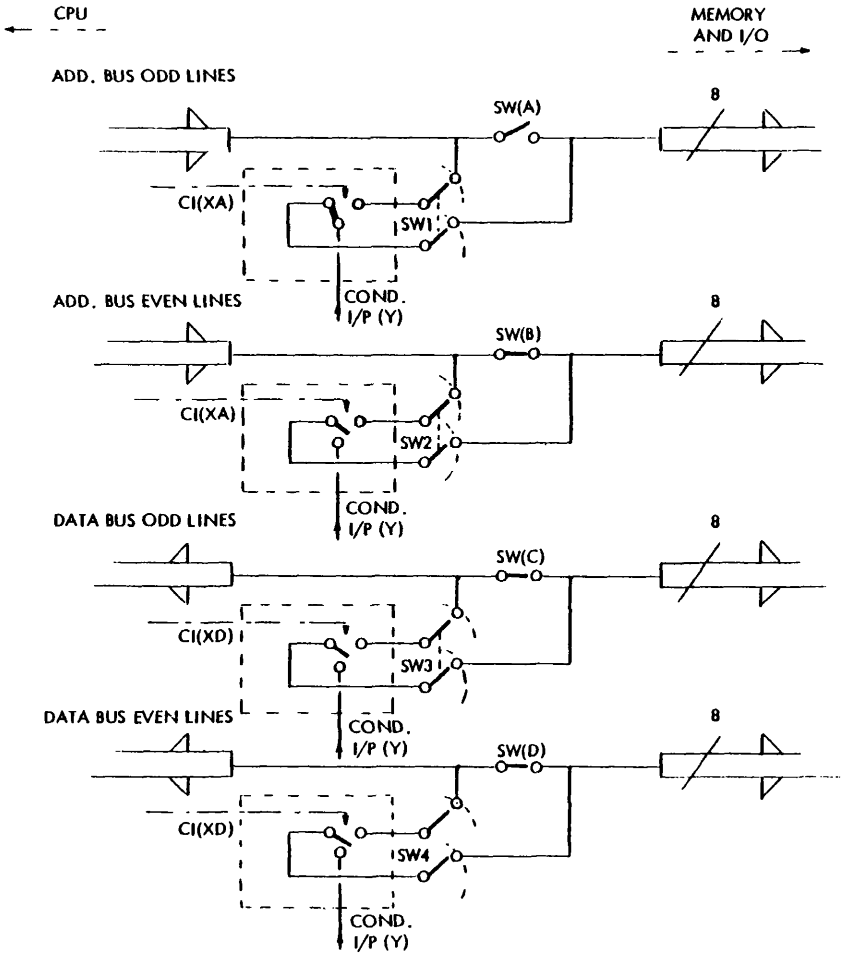


Fig. 6. 18. Control/Condition Input Circuitry.



NOTES

1. Fault Injection Mechanism

The fault injection mechanism consists of four groups of 8 lines as follows:

Group A	:	Odd Address lines
Group B	:	Even Address lines
Group C	:	Odd Data lines
Group D	:	Even Data lines

Each group selectable to the Fault Injection Logic.

2. Fault Selection Switches

All lines in each group can be individually selected for fault injection, up to a maximum of one line per group.

This is shown diagrammatically in the figure where SW(A) to SW(D) represent 4 x 8 single pole switches, connected as follows:

Normally Closed	:	Unfaulted Condition
Open	:	Faulted Condition

Switches SW1 to SW4 enable the faulted line to be connected via the fault injection logic.

Position of switches as indicated in figure represent the selection of an Odd Address fault.

3. Legend

CI( )	Control Input ( )
COND. I/P (Y)	Condition Input (Y)

Fig. 6.19. Fault Injection Switching Arrangement.

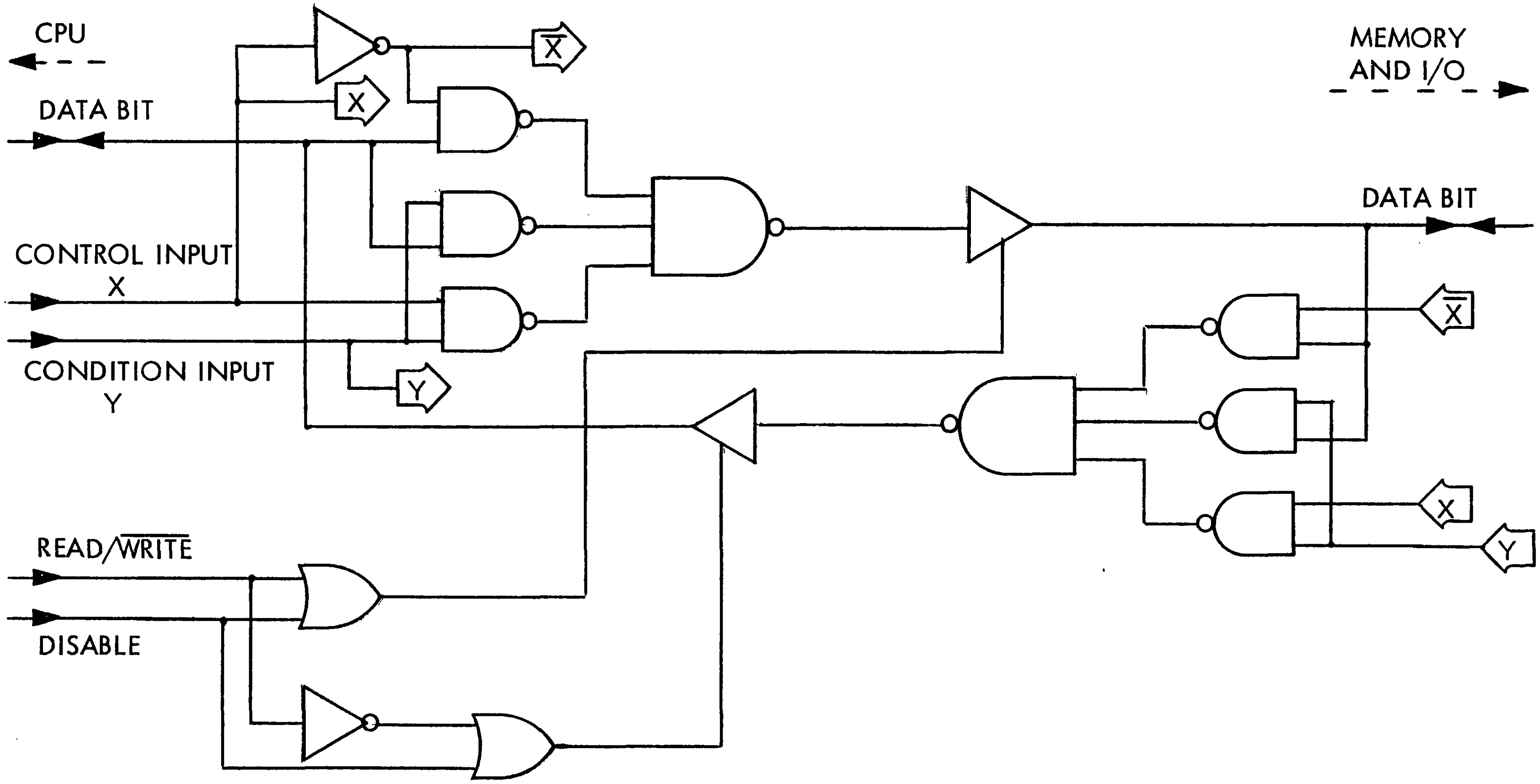


Fig. 6.20. Implementation of Data Fault Logic.

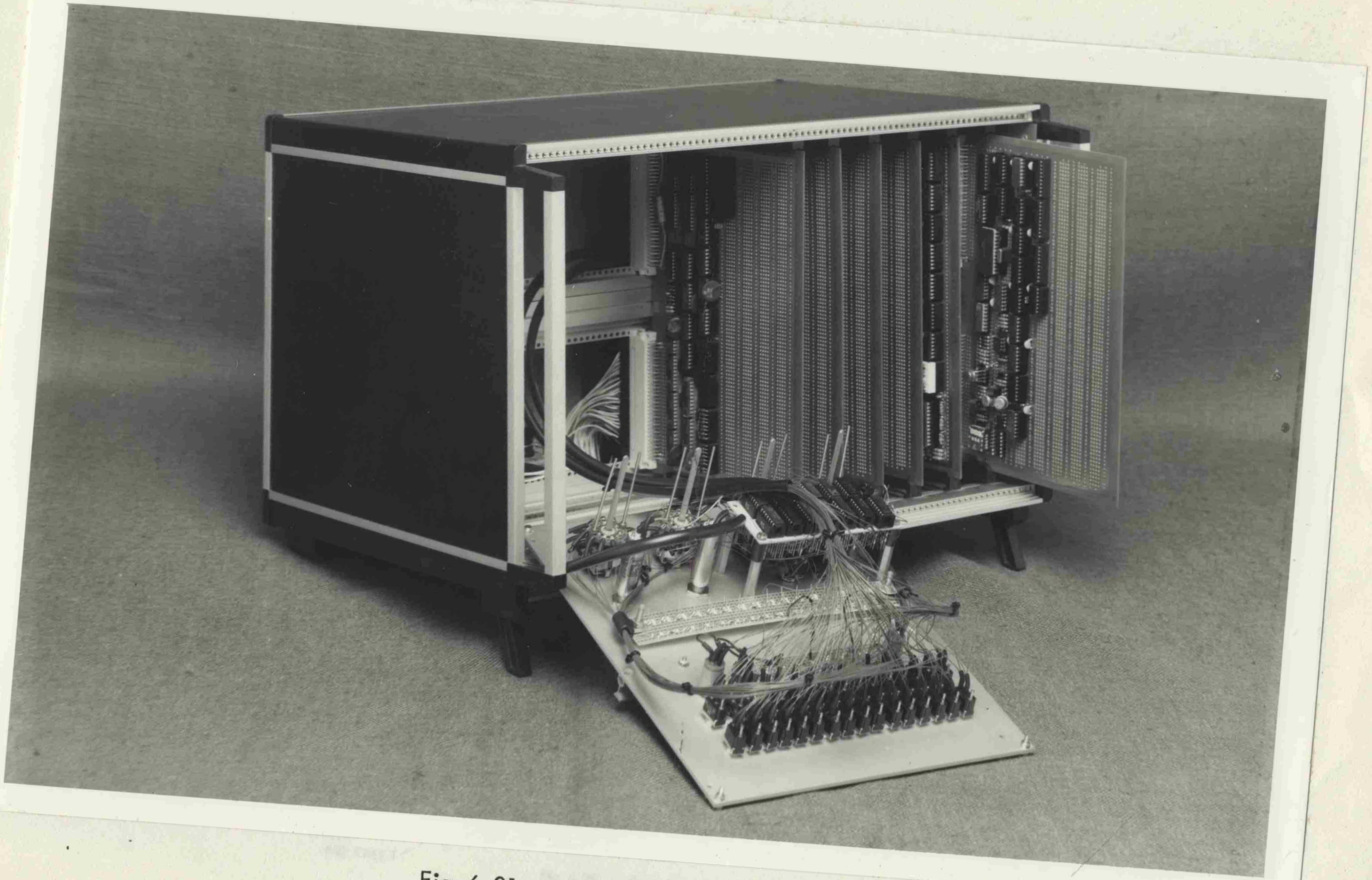


Fig.6.21. Fault Injection Logic.

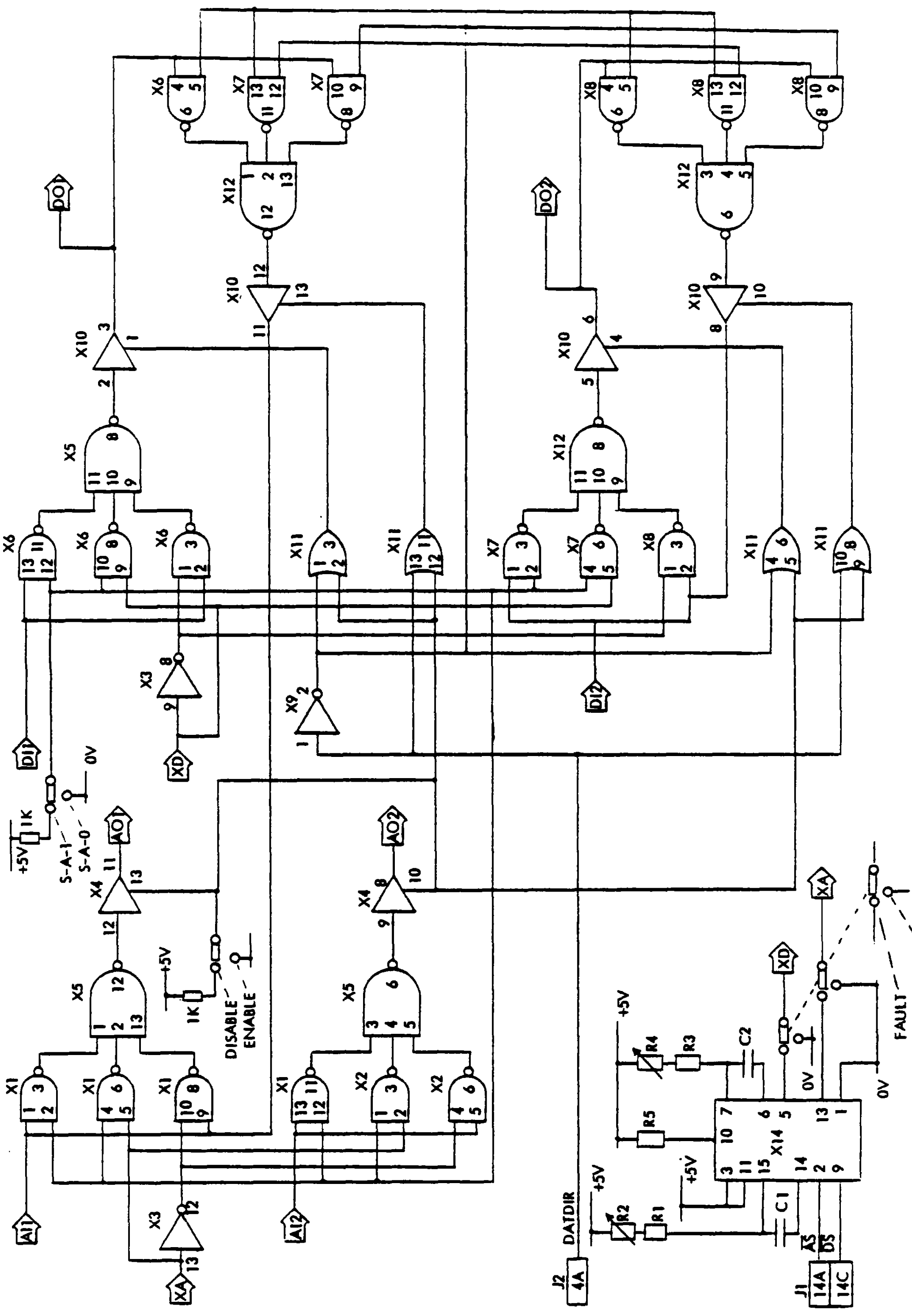


Fig. 6.22. Fault Injection Logic.

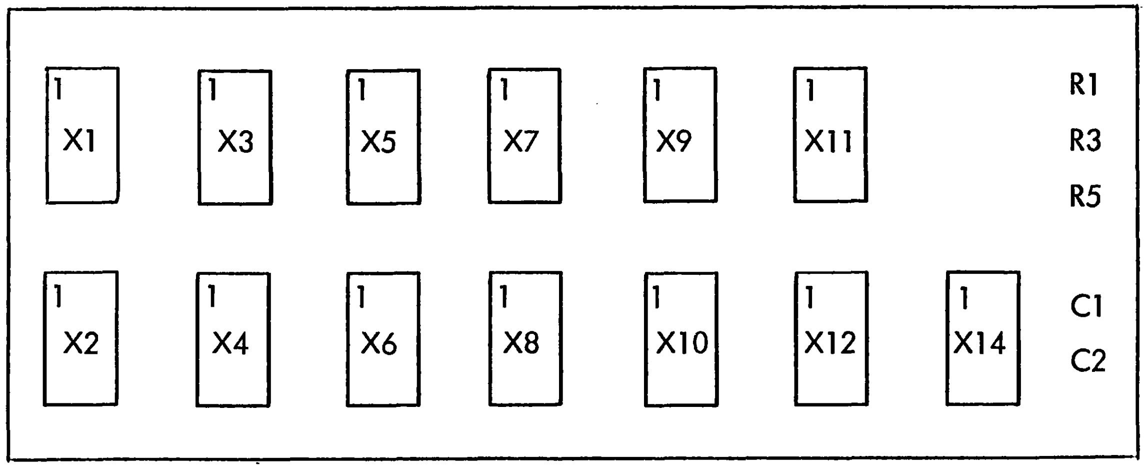


Fig.6.23. Layout of Fault Injection Logic.

```
LEVEL 1.        TASKING SEQUENCE     E.G.  CALL TASK A
                                           CALL TASK B
                                                .
                                                .
                                                .

LEVEL 2.        RECOVERY BLOCK       I.E.  ENSURE T
                                           BY     P
                                           ELSE   Q
                                           ELSE ERROR
                                              .
                                              .
                                              .

LEVEL 3.        CODE FOR EACH PRIMARY AND SECONDARY
                ROUTINE  AND ACCEPTANCE TEST.
```

Fig.7.1.   Three Level Structure.

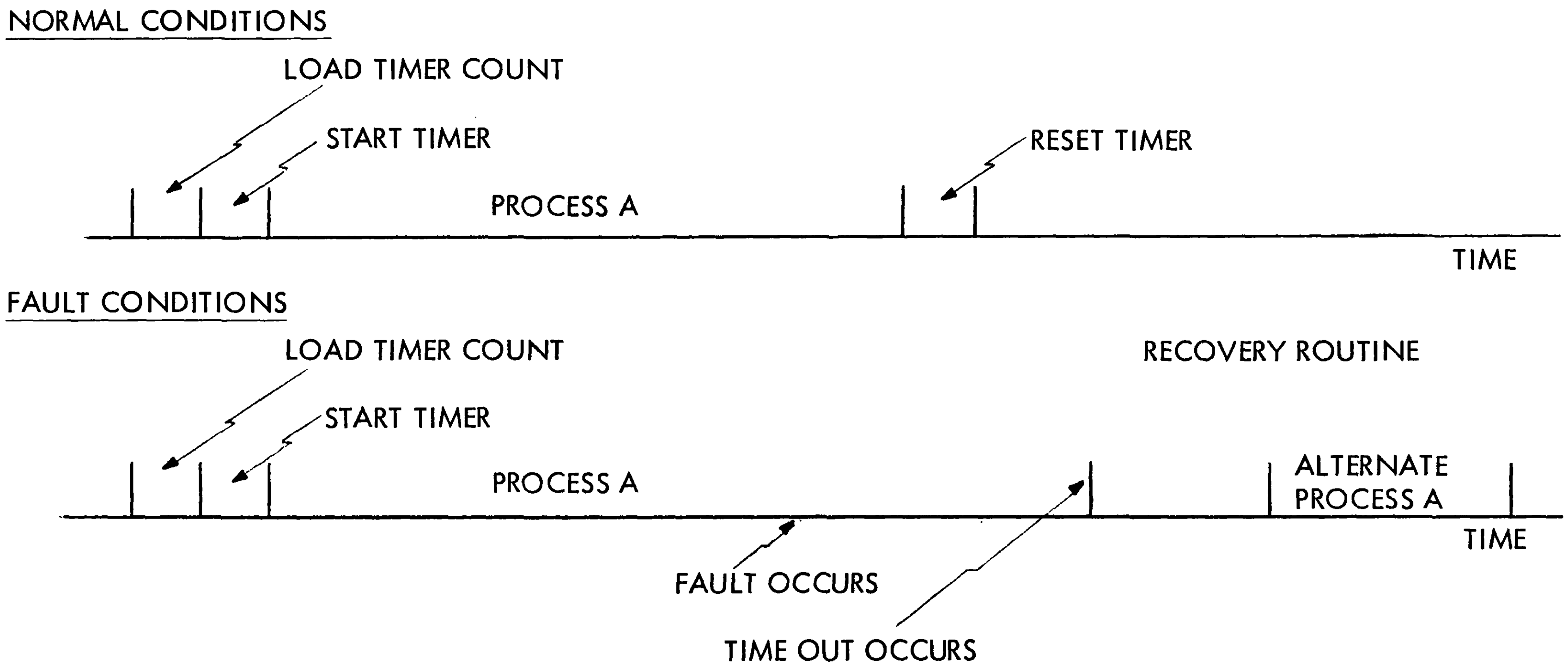


Fig.7.2. Schematic of Watchdog Timer.

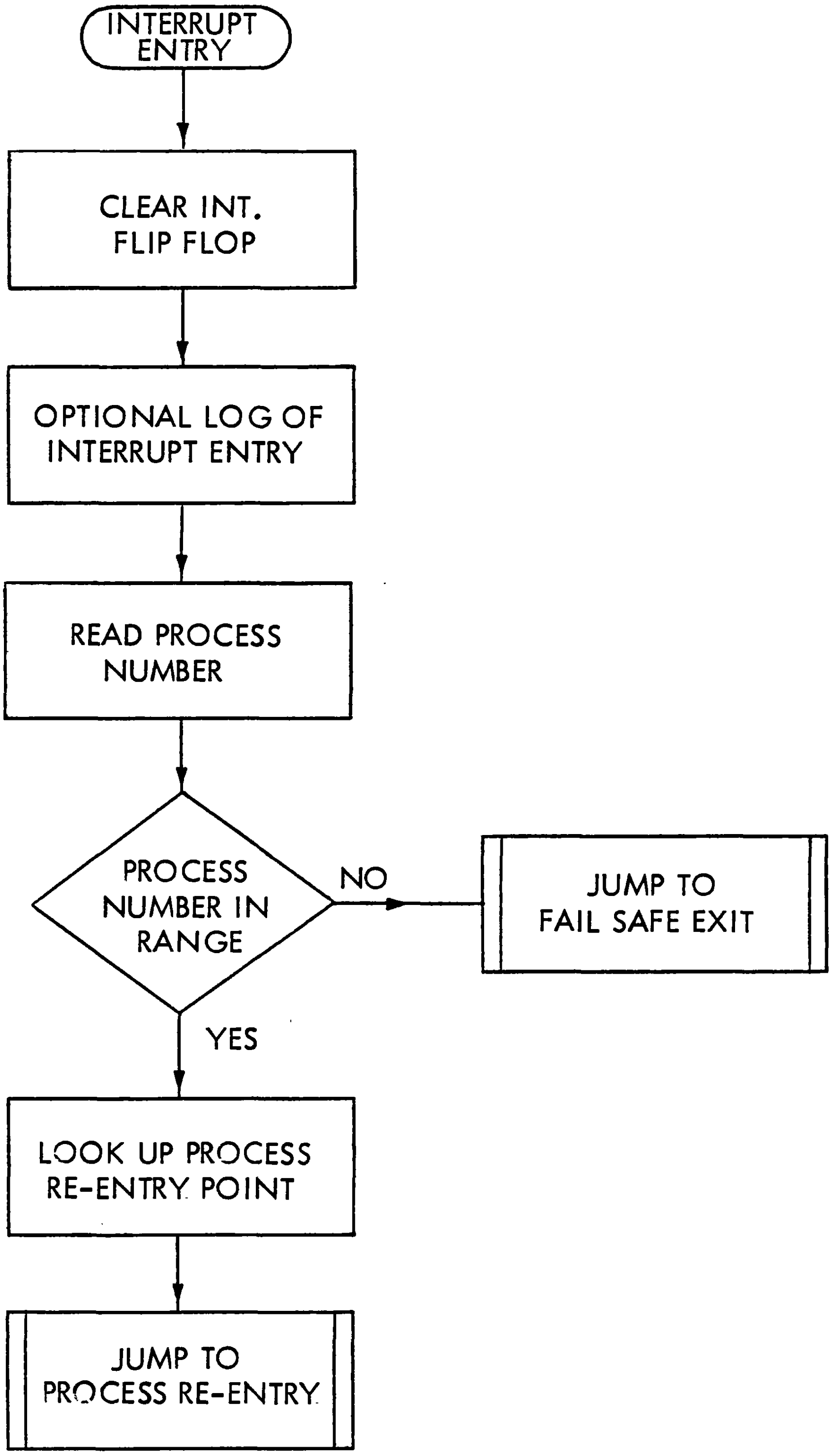


Fig.7.3. Recovery Interrupt Service Routine.

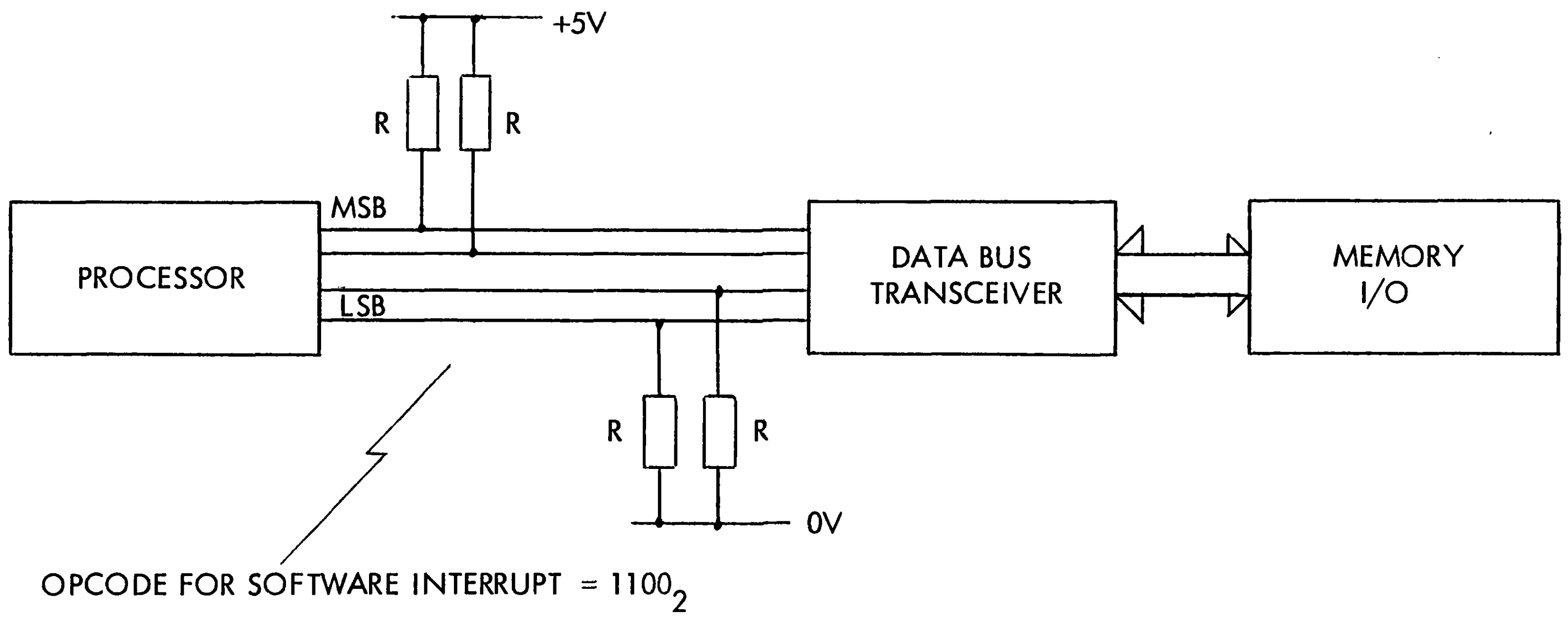


Fig.7.4. Default Data Bus.

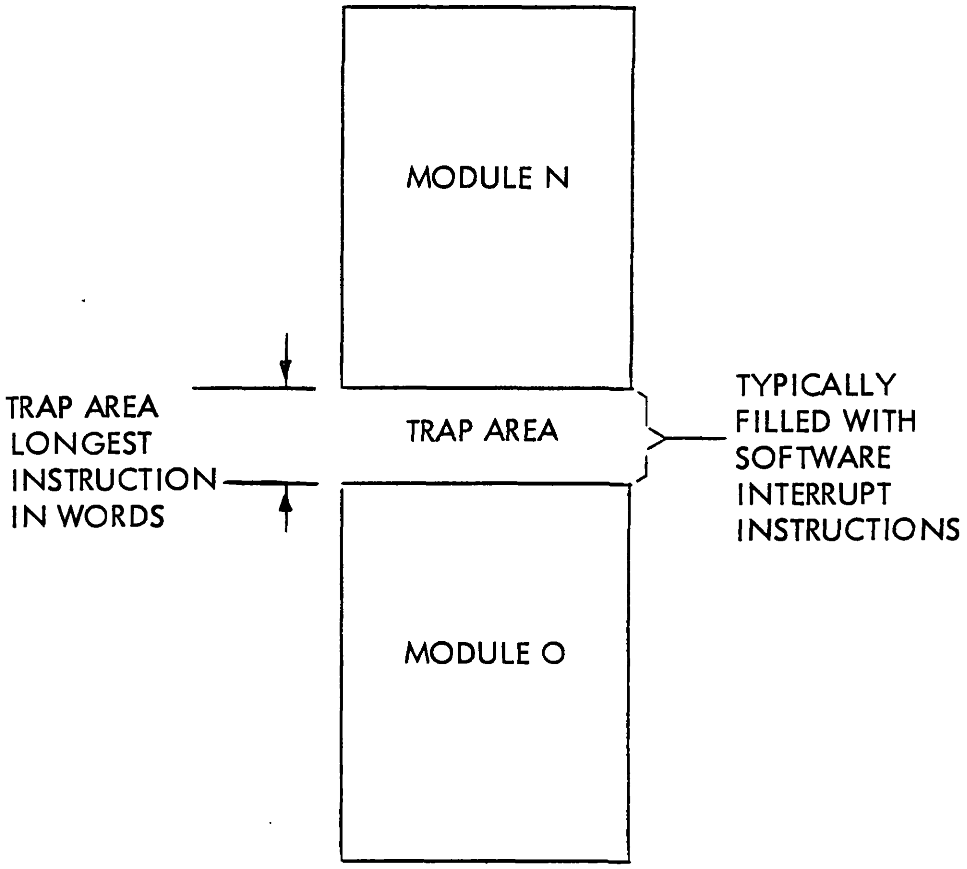
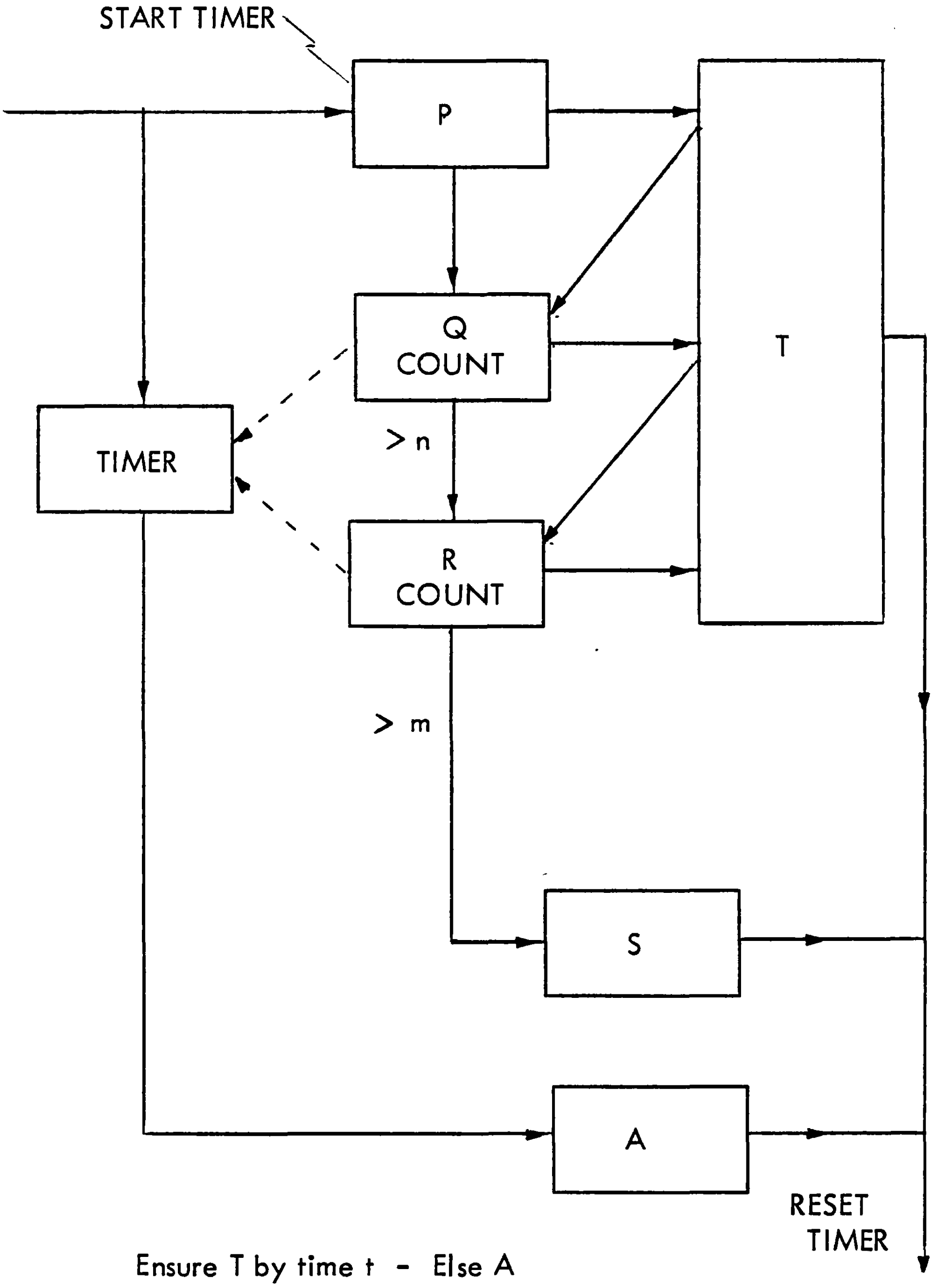

MODULE N
TRAP AREA
MODULE O
TRAP AREA LONGEST INSTRUCTION IN WORDS
TYPICALLY FILLED WITH SOFTWARE INTERRUPT INSTRUCTIONS

Fig.7.5. Schematic of Trap Area.



```
Ensure T by time t  -  Else A
        By P
        Else Q if Q has not been used n times
        Else R if R has not been used m times
Else S
Else Error
```

Fig.7.6.   Generalised Form of Recovery Block.

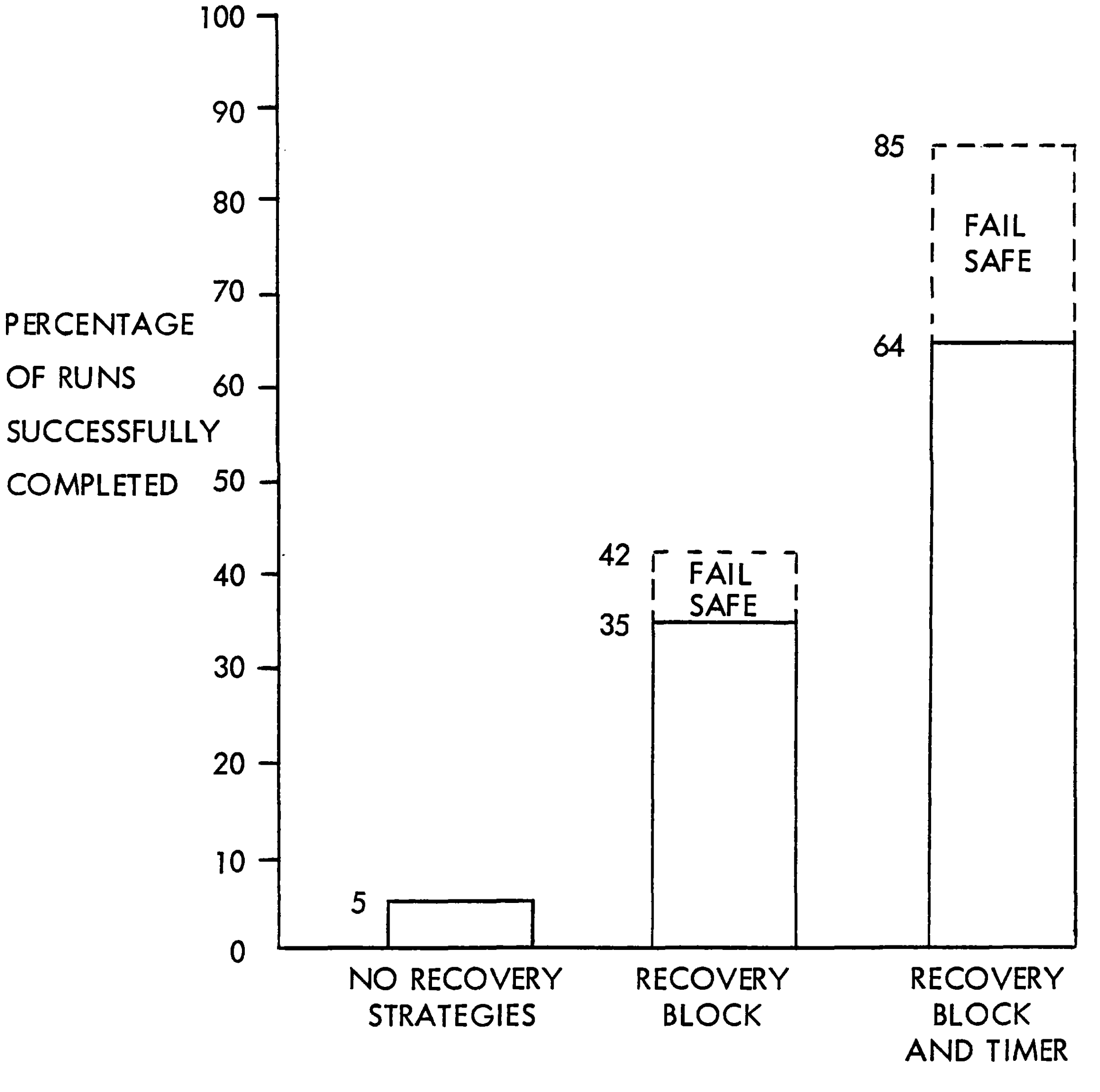


Fig.8.1. System Availability Related to Recovery Strategies

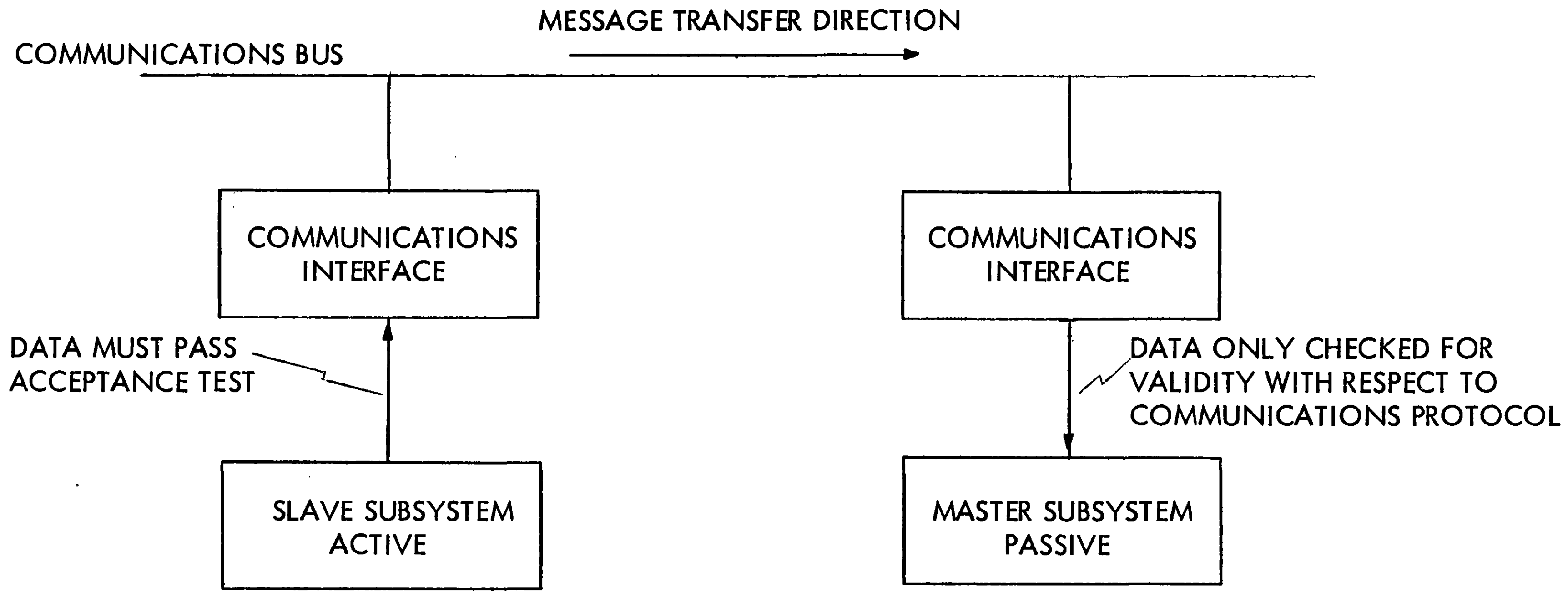
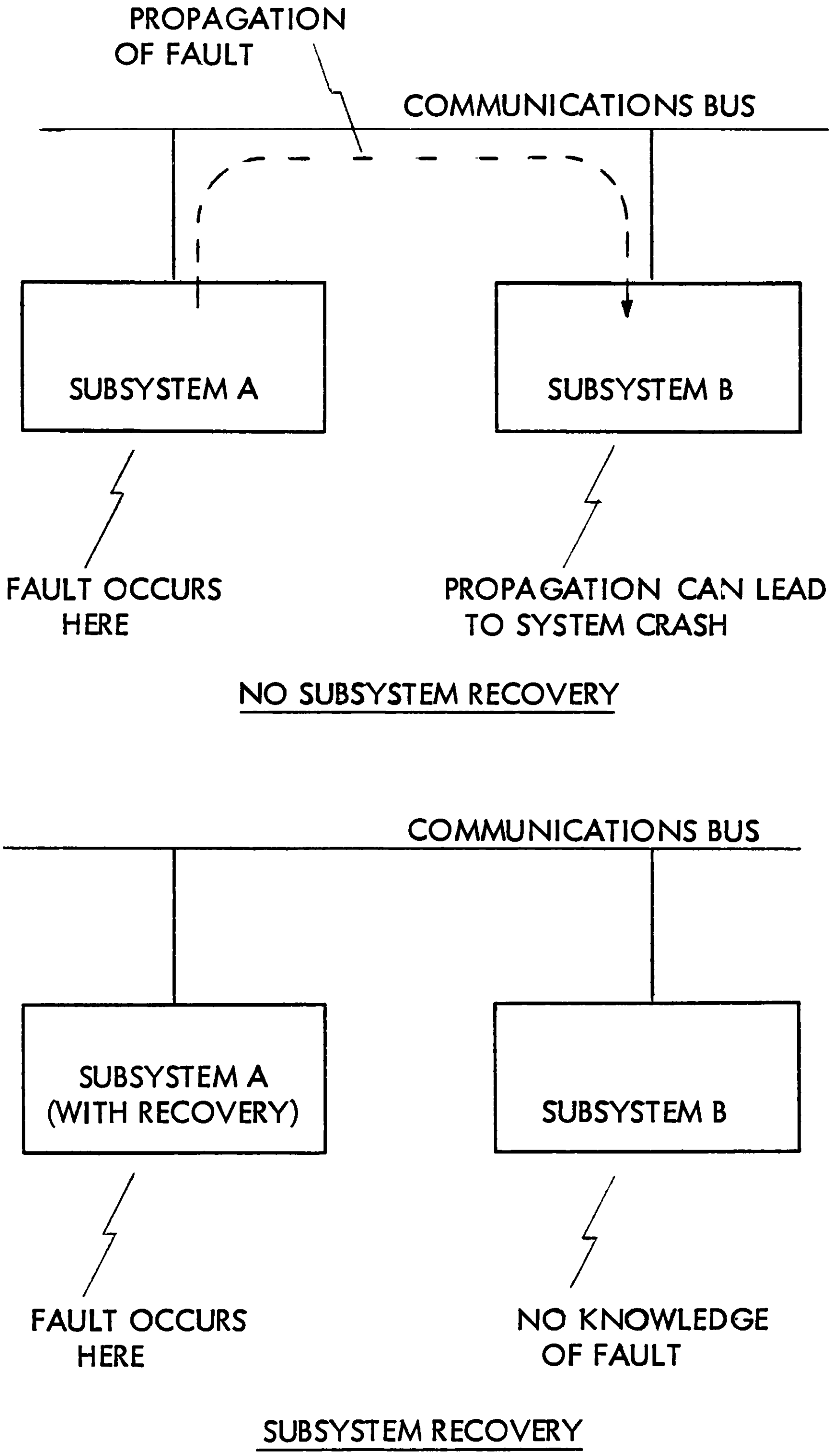

MESSAGE TRANSFER DIRECTION
COMMUNICATIONS BUS
COMMUNICATIONS INTERFACE
COMMUNICATIONS INTERFACE
DATA MUST PASS ACCEPTANCE TEST
DATA ONLY CHECKED FOR VALIDITY WITH RESPECT TO COMMUNICATIONS PROTOCOL
SLAVE SUBSYSTEM ACTIVE
MASTER SUBSYSTEM PASSIVE

Fig.9.1. Design Philosophy for Inter Processor Communication.



NO SUBSYSTEM RECOVERY

SUBSYSTEM RECOVERY

Fig.9.2. Local Recovery Strategy.

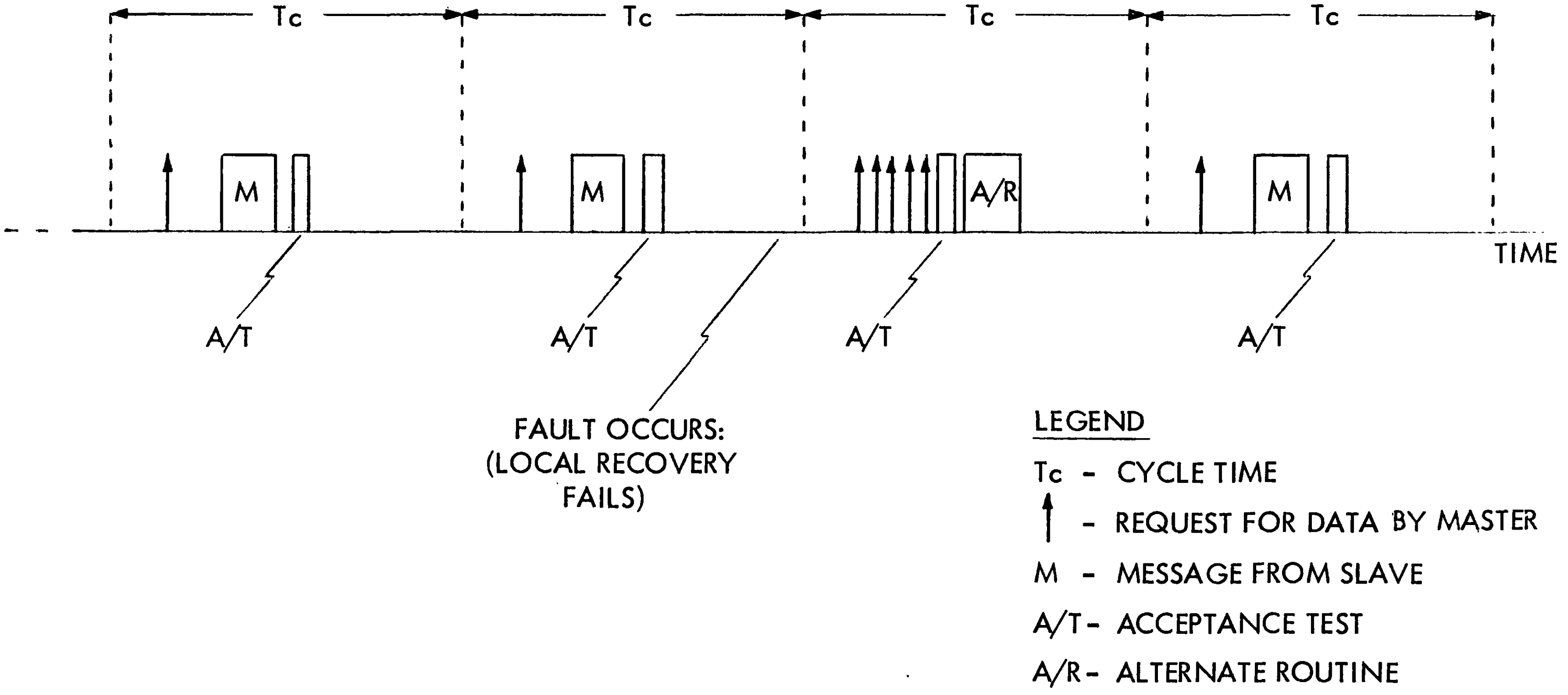


Fig.9.3. Global Recovery Strategy.

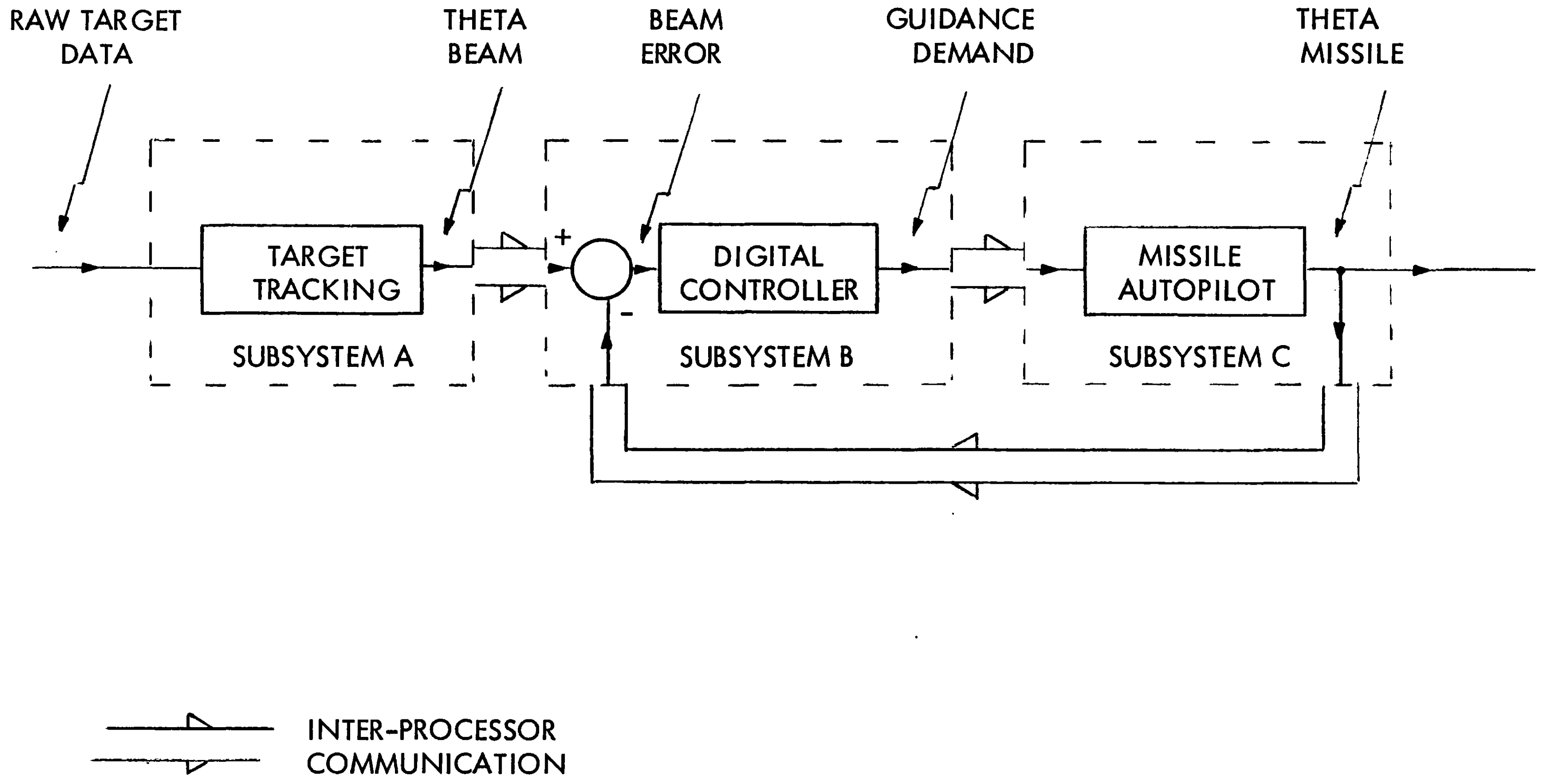


Fig.9.4. Separation of Functions in Distributed System.

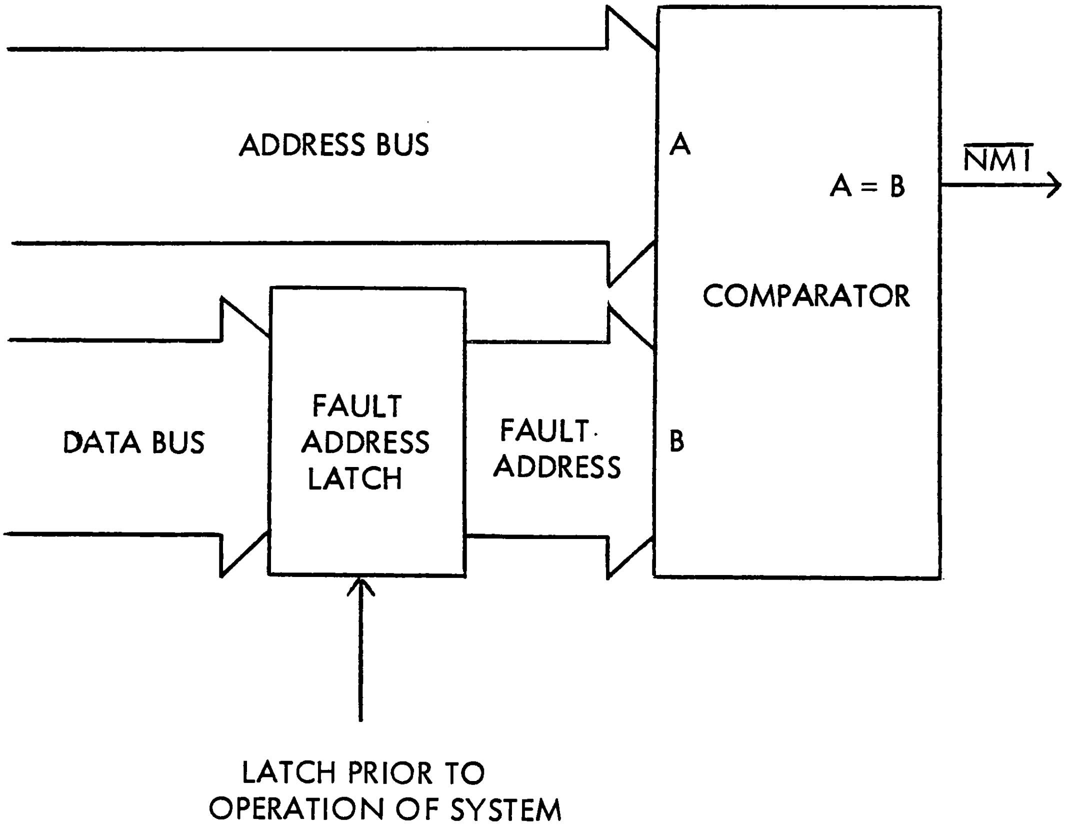


Fig.9.5. Schematic of Real Time Fault Injection Mechanism.

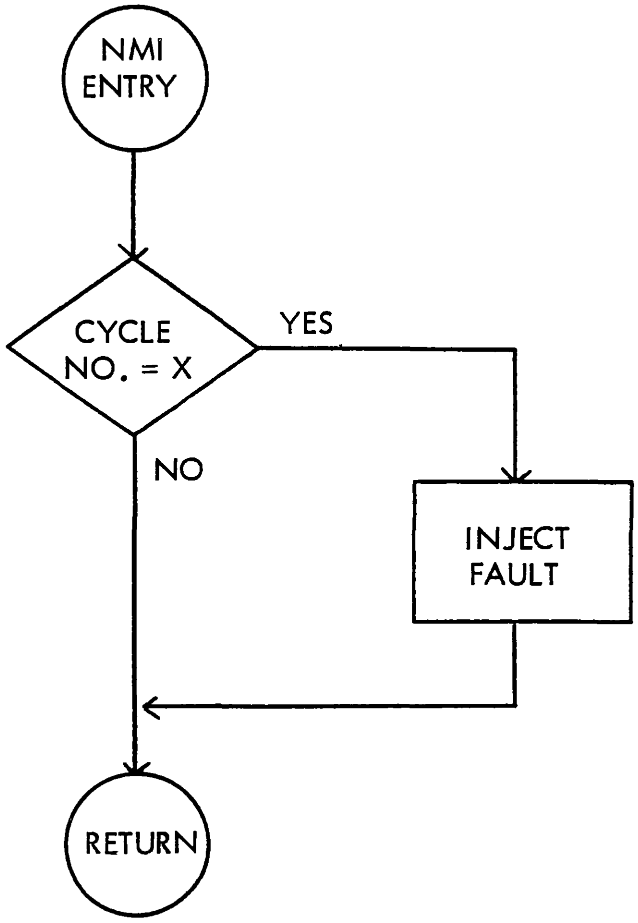
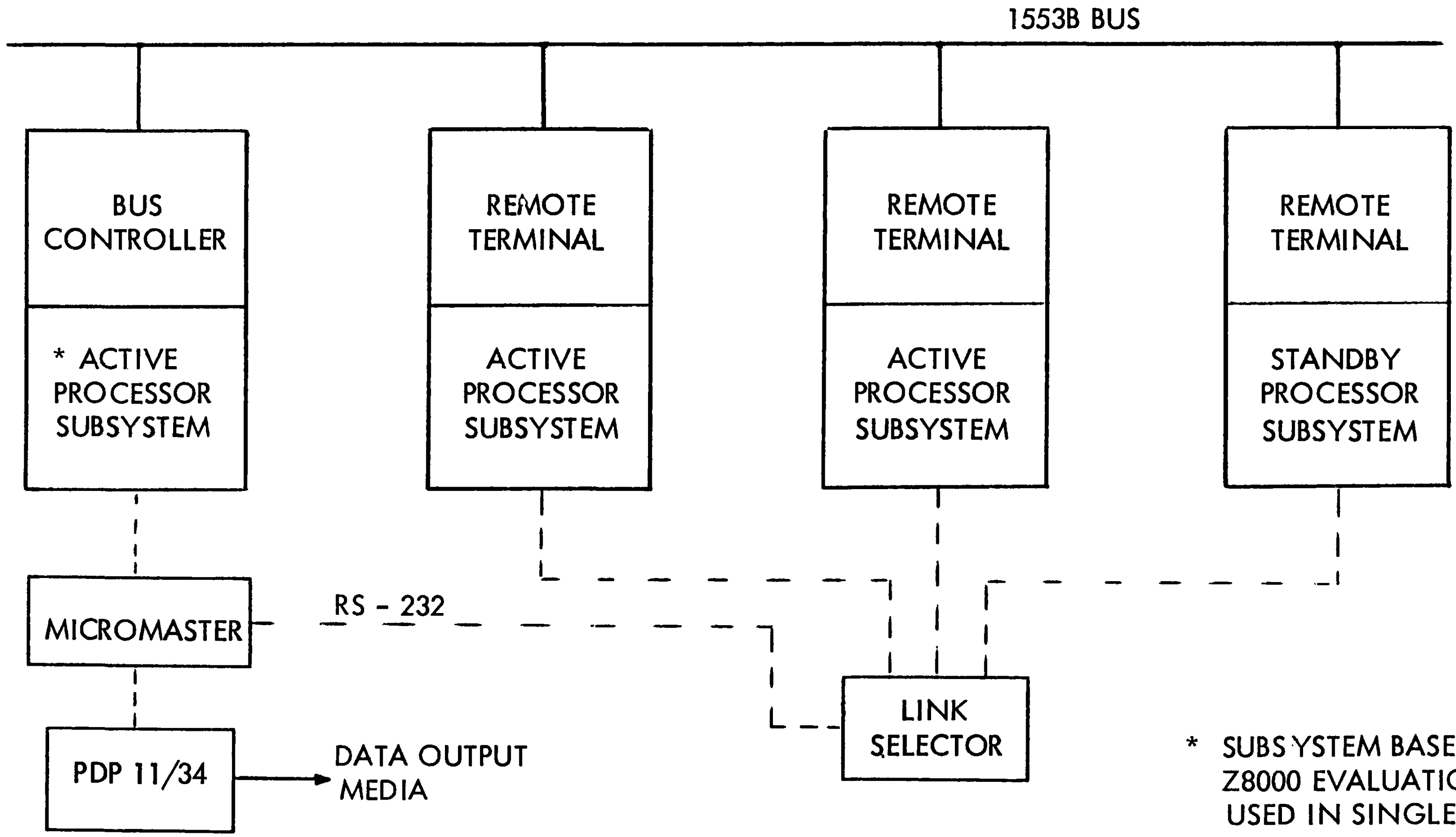


Fig.9.6. Specific Cycle Fault Injection.



* SUBSYSTEM BASED AROUND Z8000 EVALUATION CARD AS USED IN SINGLE PROCESSOR STUDY

Fig. 10.1. Overall Systems Diagram for Distributed Processing System.

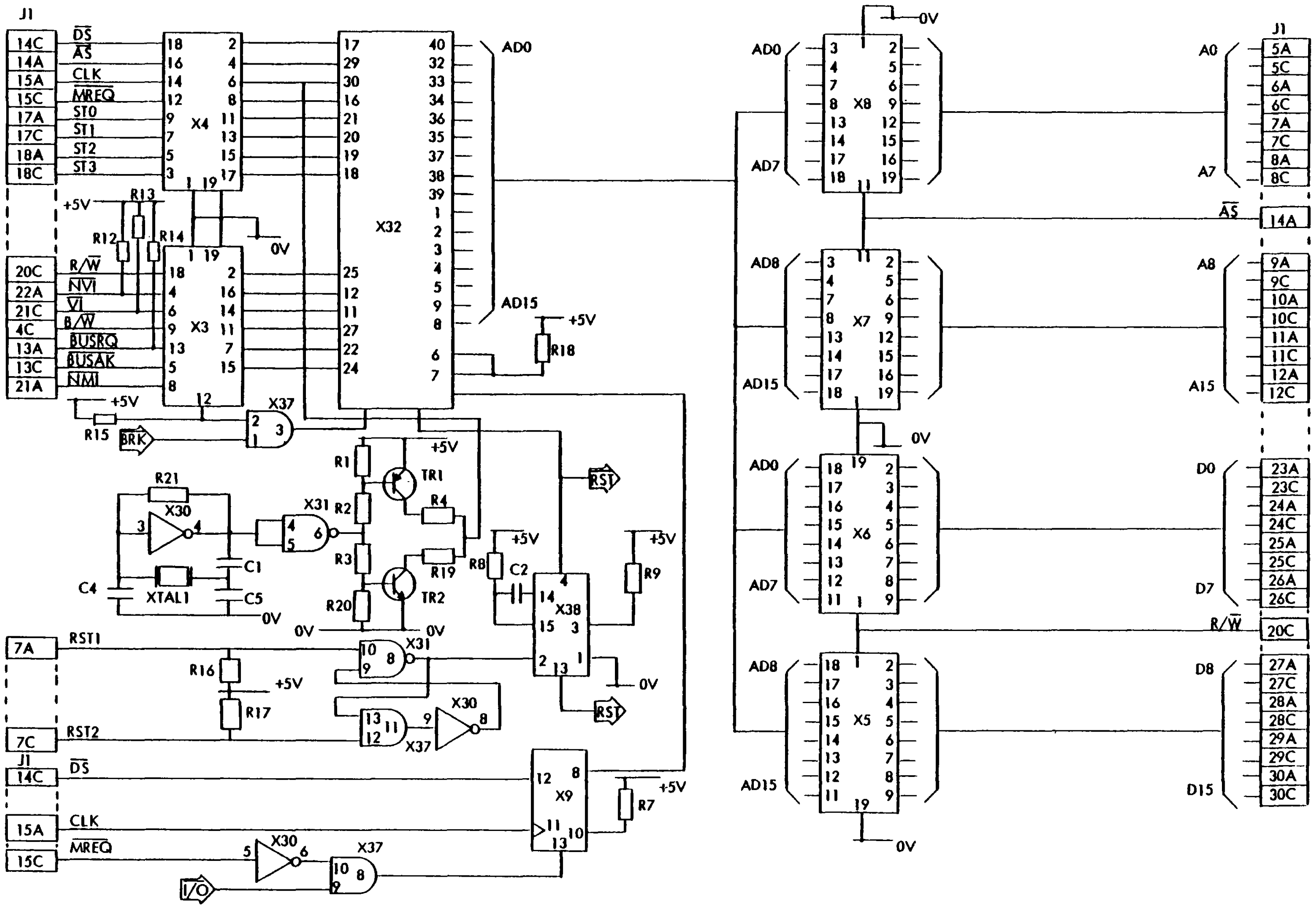


Fig. 10.2.a. Central Processing Unit.

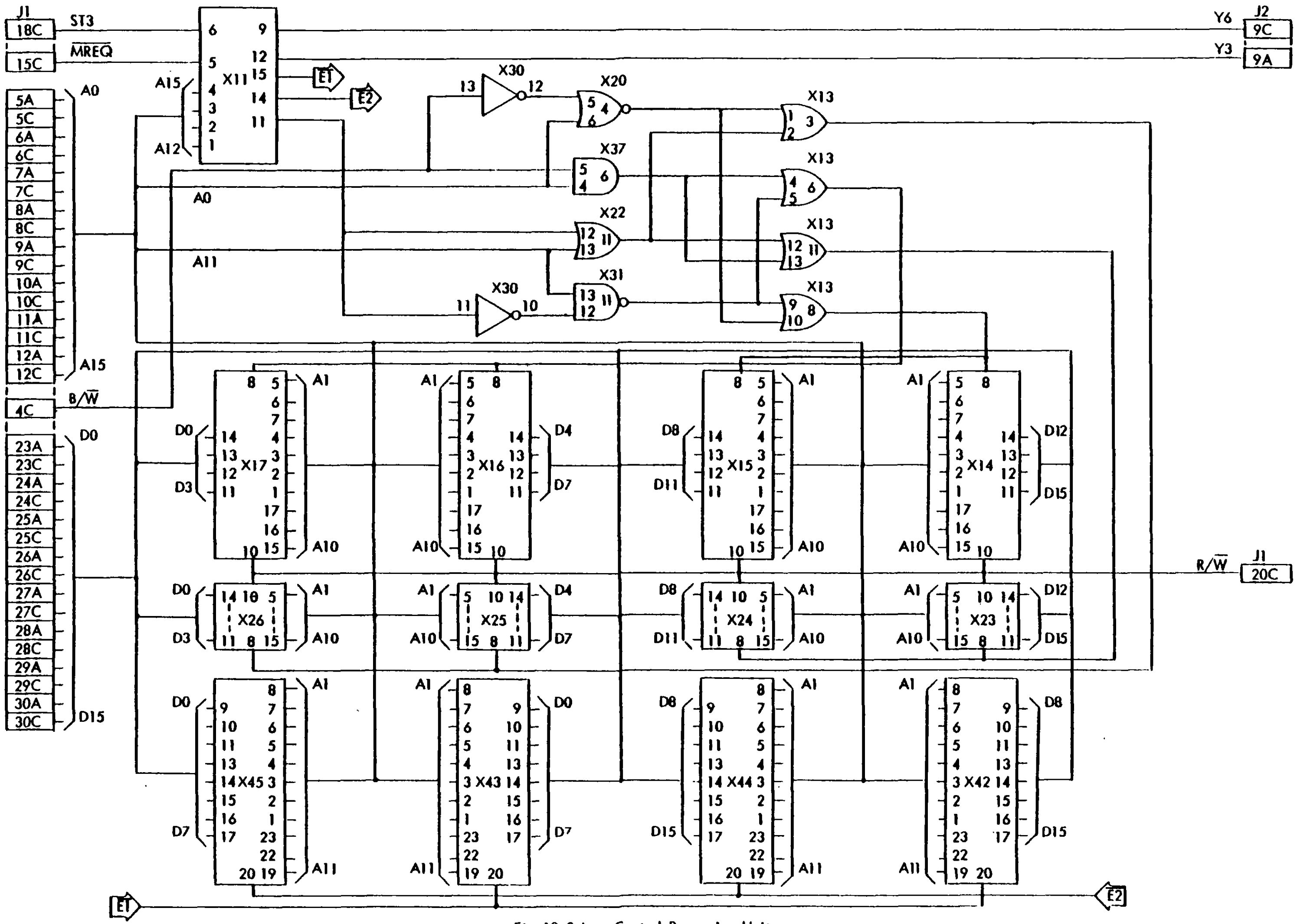


Fig. 10.2.b. Central Processing Unit

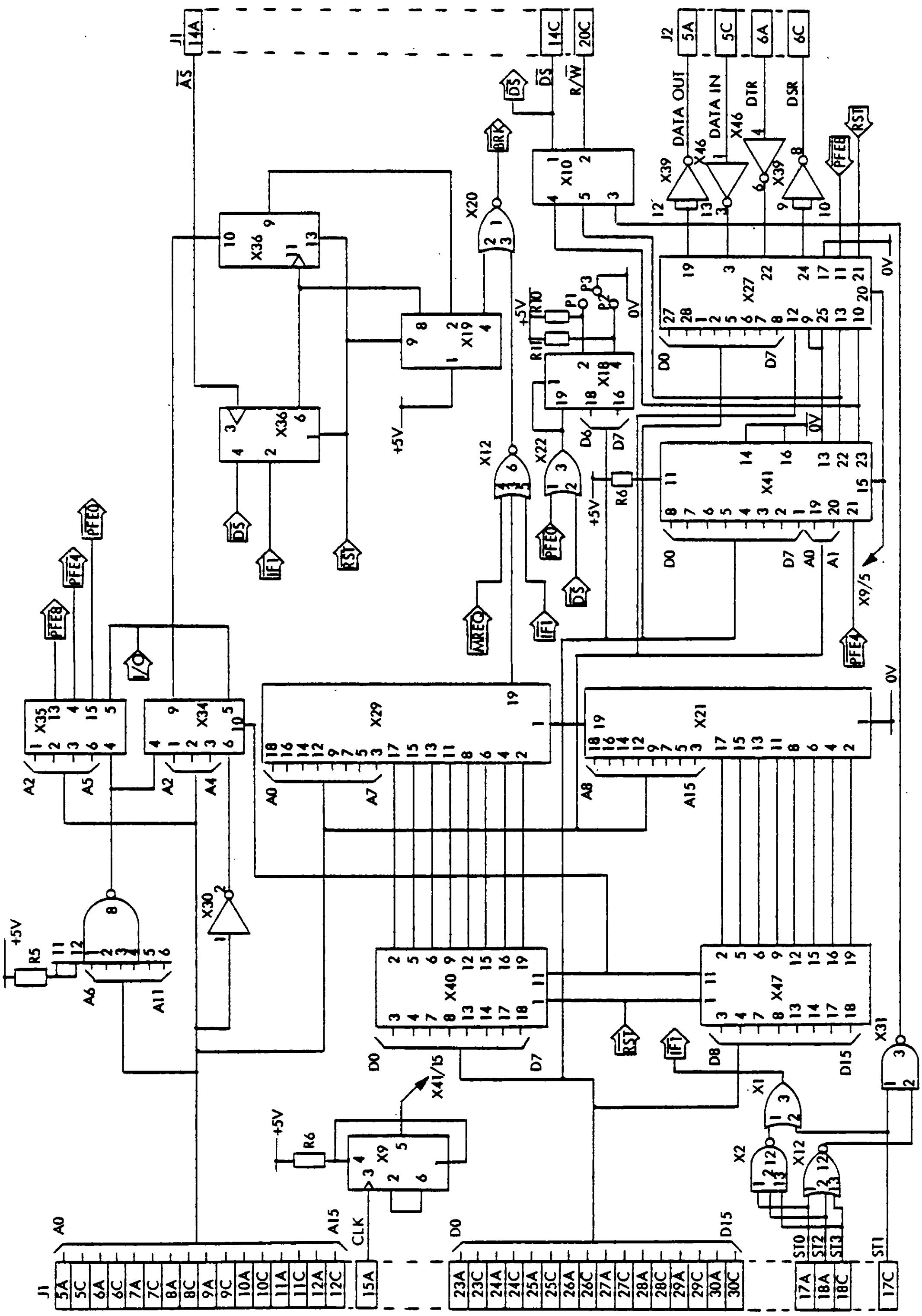


Fig. 10.2.c. Central Processing Unit.

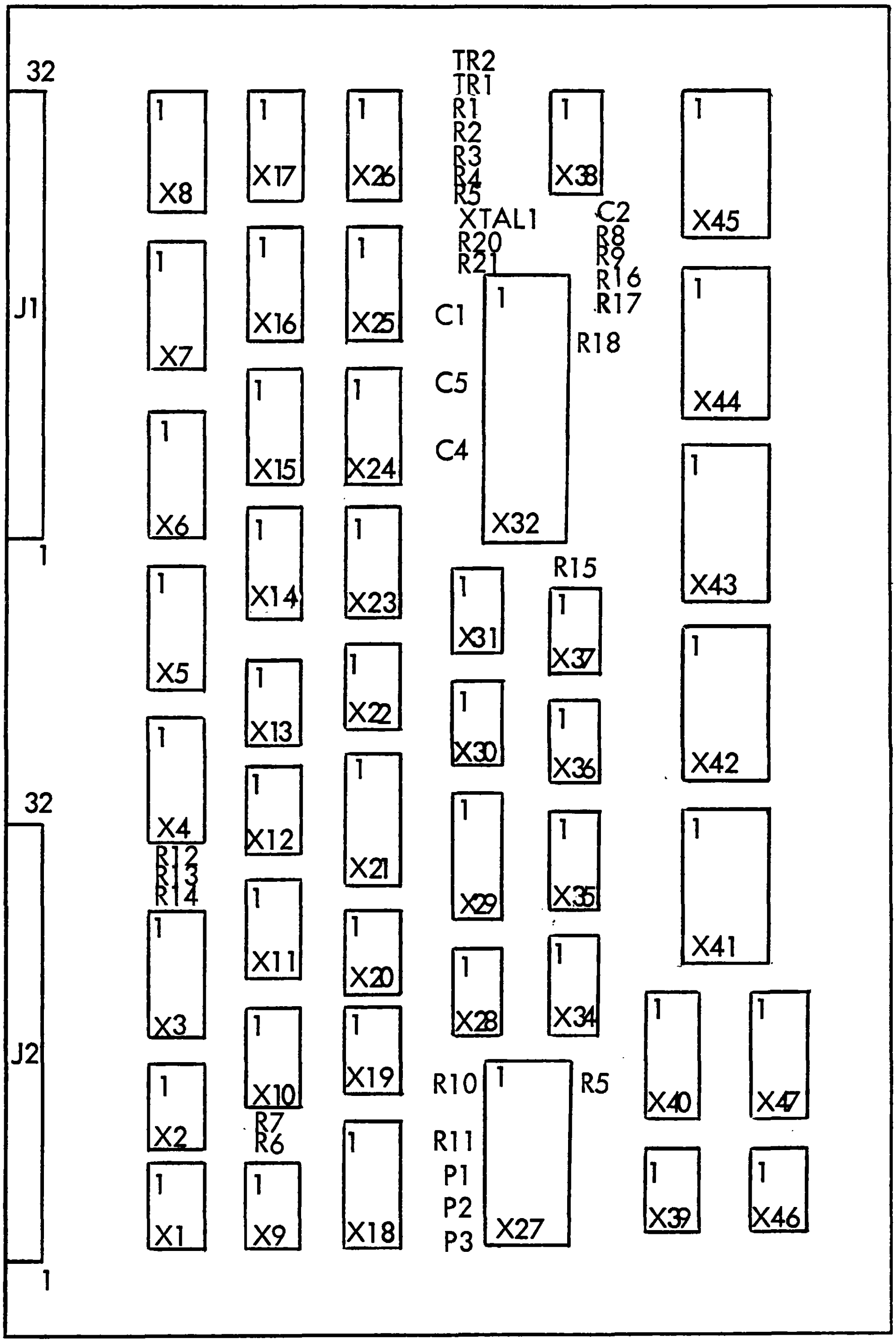


Fig. 10.3. Layout of Central Processing Unit.

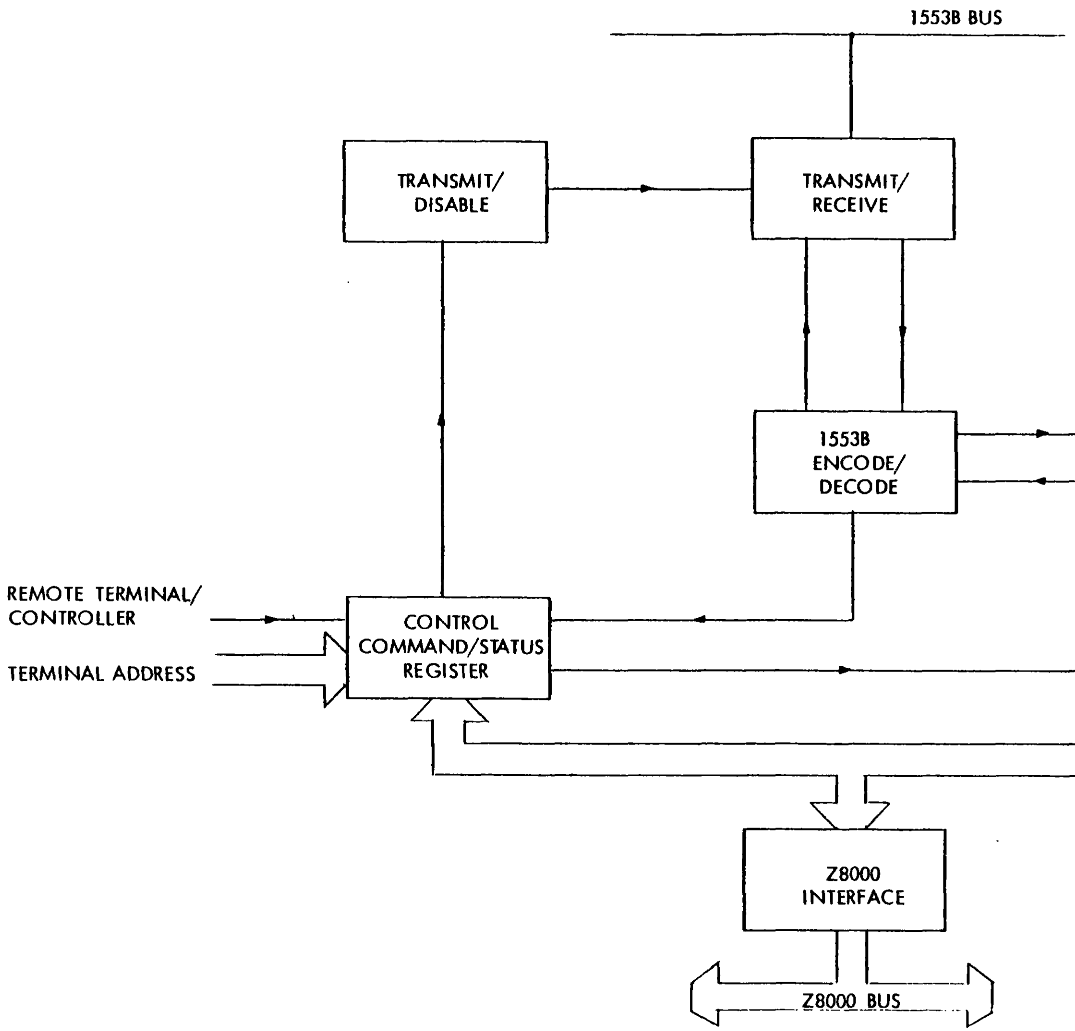

1553B BUS
TRANSMIT/ DISABLE
TRANSMIT/ RECEIVE
1553B ENCODE/ DECODE
SERIAL/PARALLEL CONVERSION
REMOTE TERMINAL/ CONTROLLER
TERMINAL ADDRESS
CONTROL COMMAND/STATUS REGISTER
MESSAGE BUFFER 32 WORD FIFO
Z8000 INTERFACE
Z8000 BUS

Fig. 10.4. Schematic of 1553B/Microprocessor Interface.

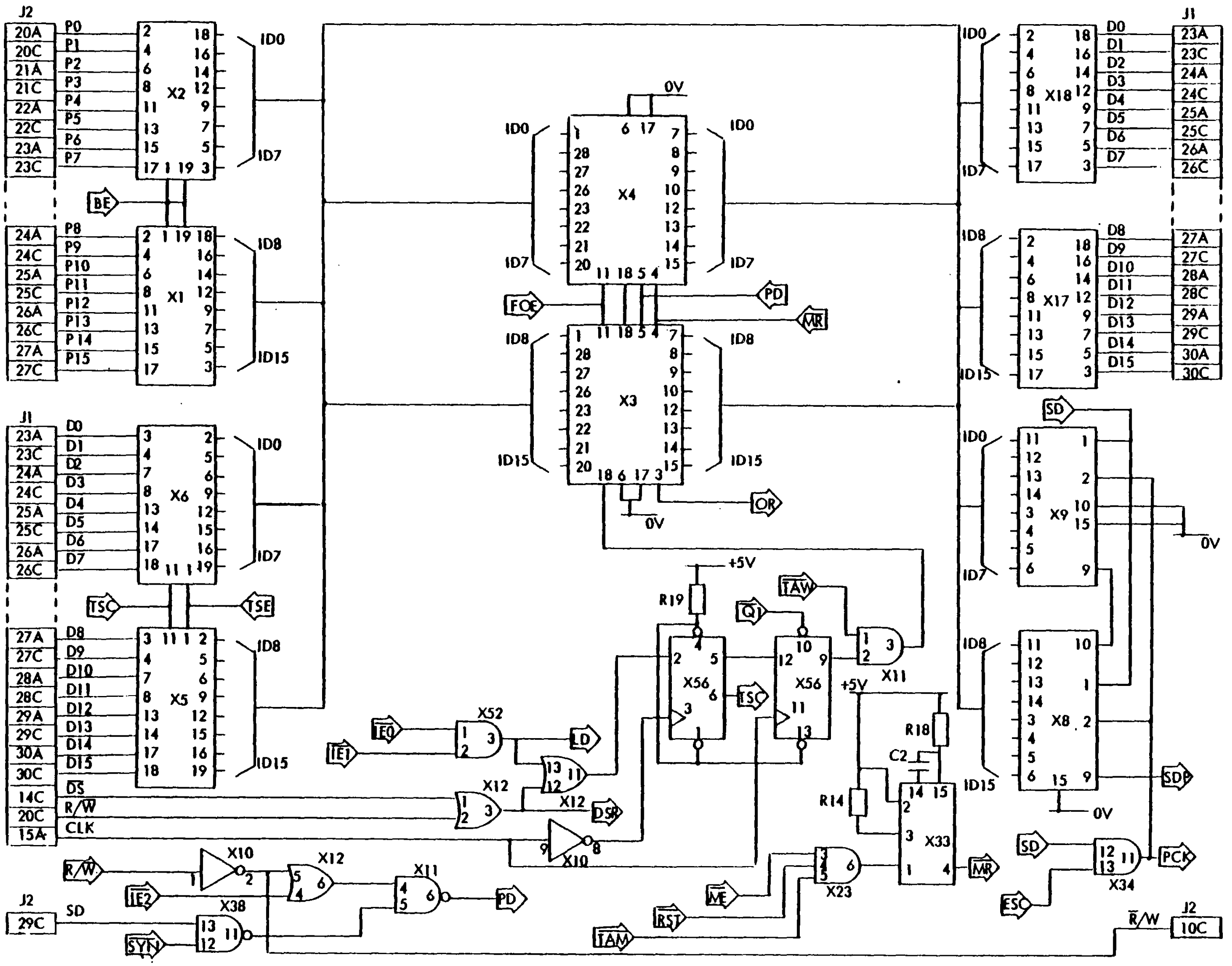


Fig.10.5.a. 1553B/Microprocessor Interface Board 1.

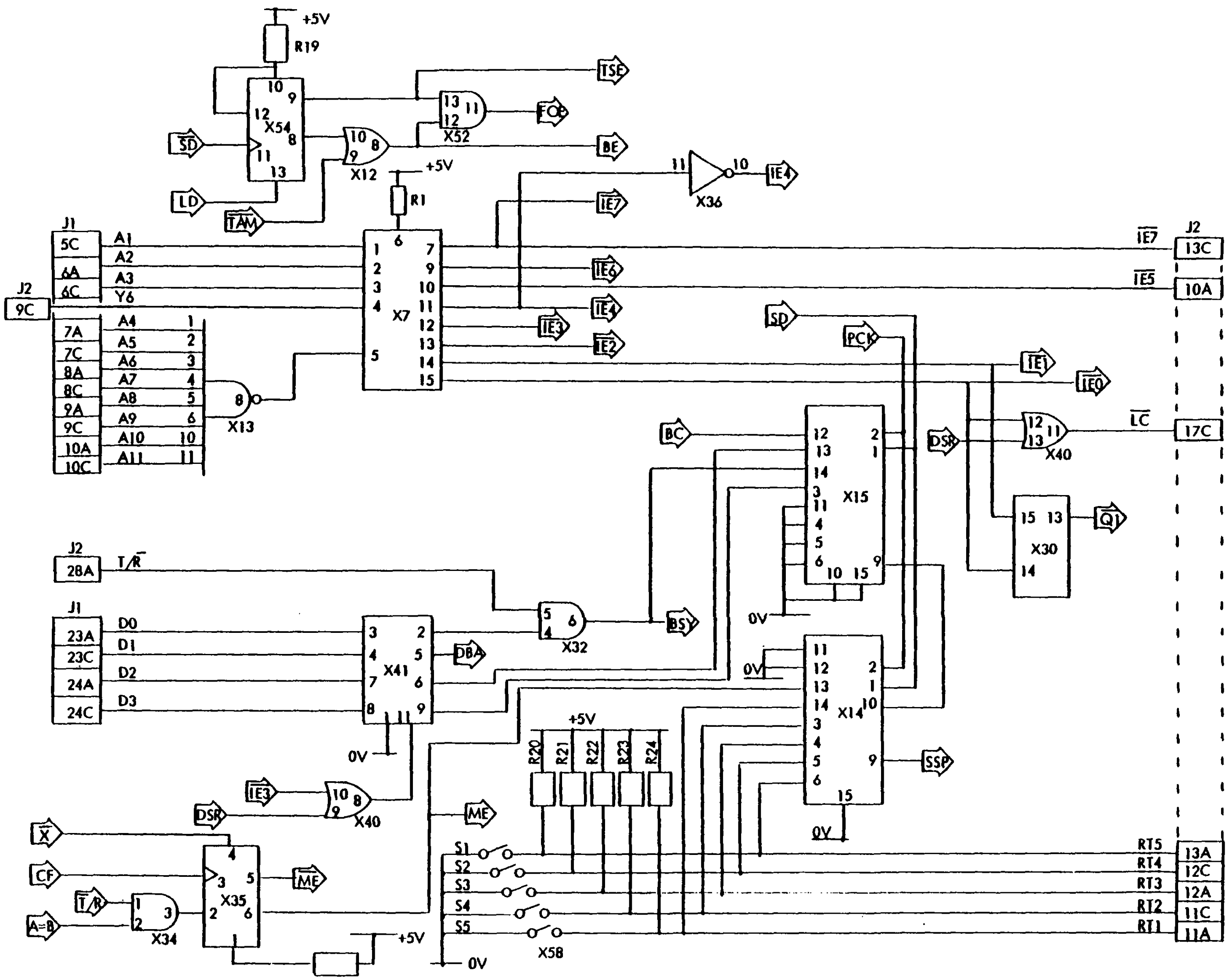


Fig. 10.5.b. 1553B/Microprocessor Interface Board 1.

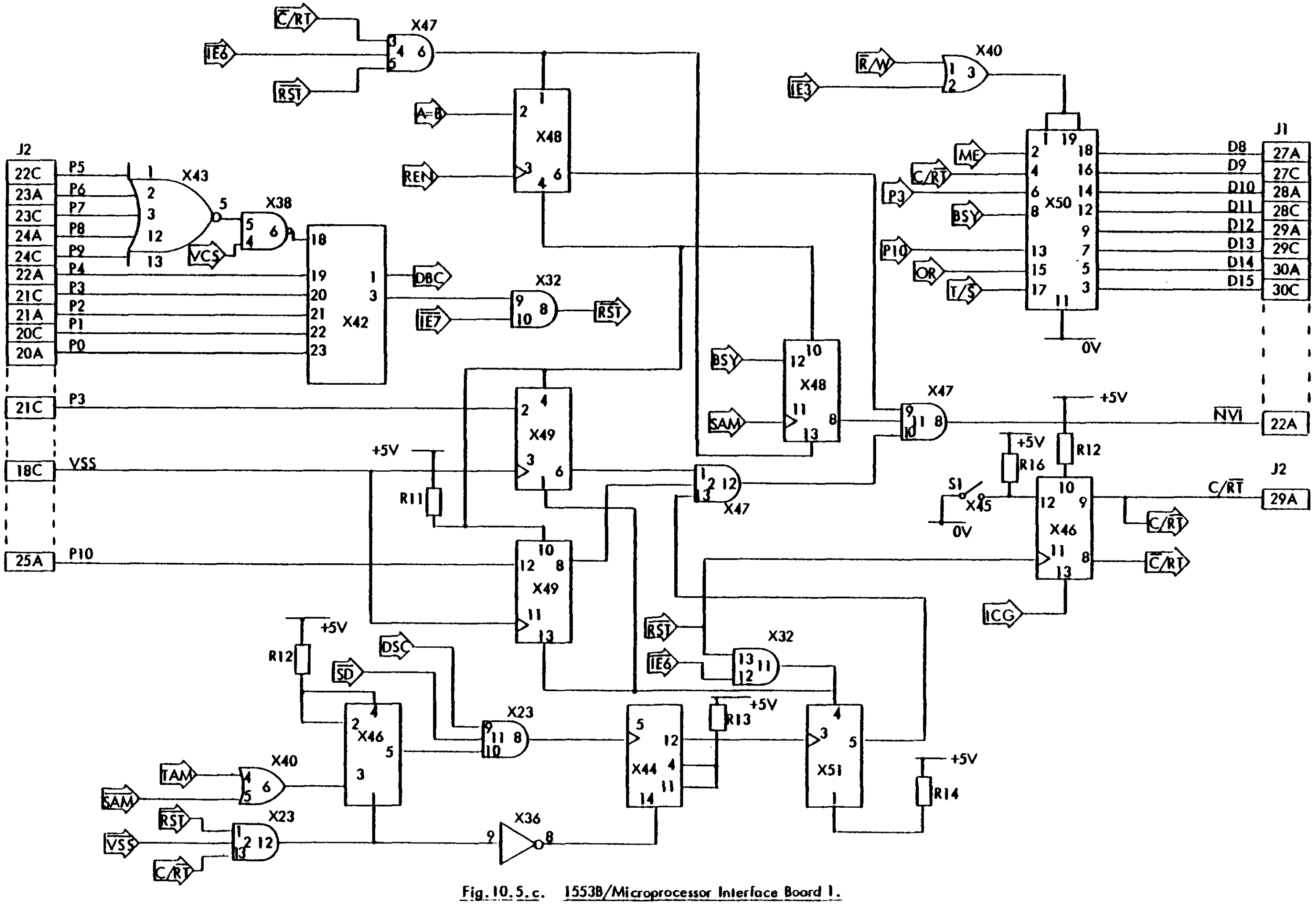


Fig. 10.5.c. 1553B/Microprocessor Interface Board 1.

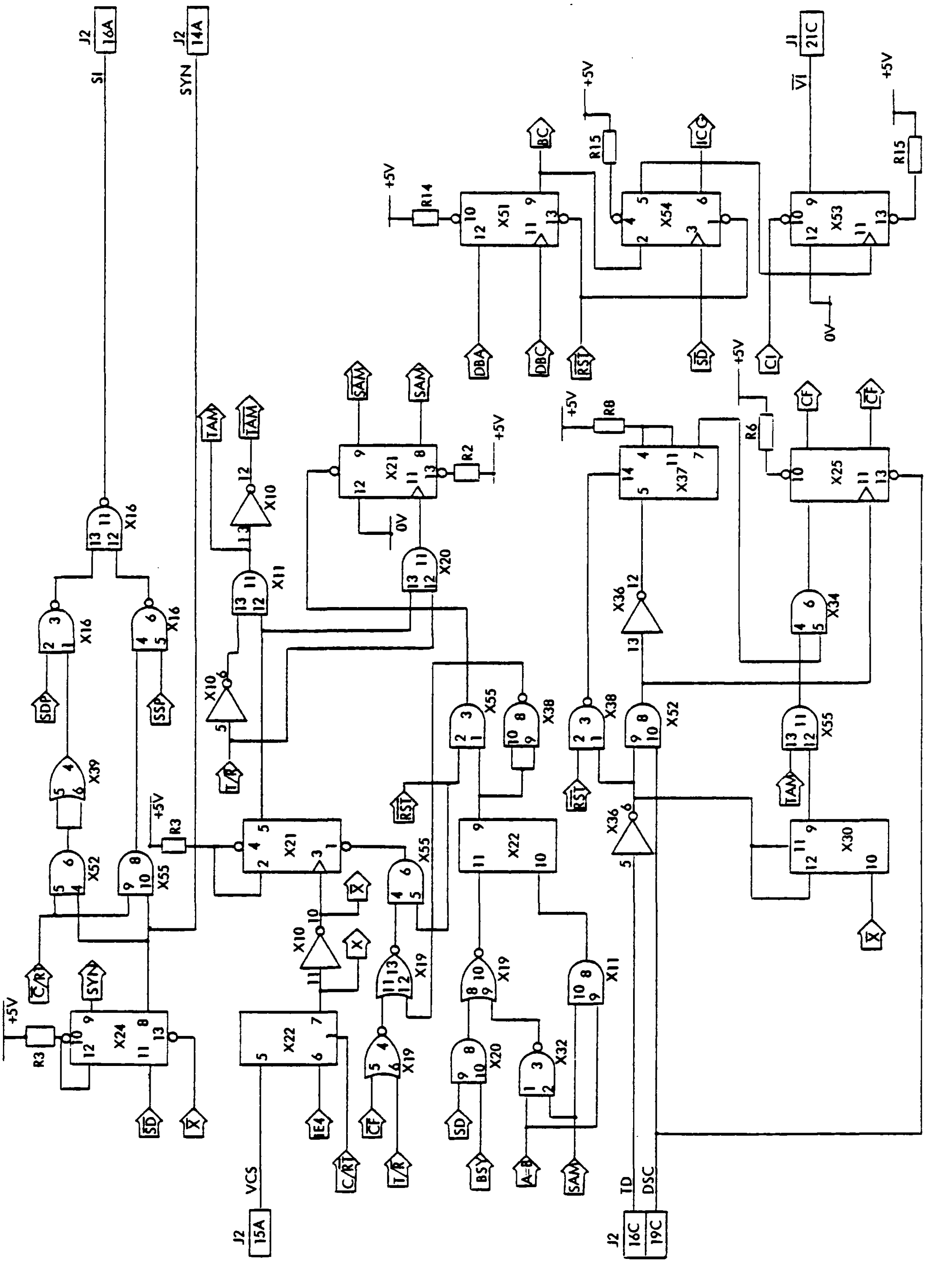


Fig.10.5.d. 1553B/Microprocessor Interface Board 1.

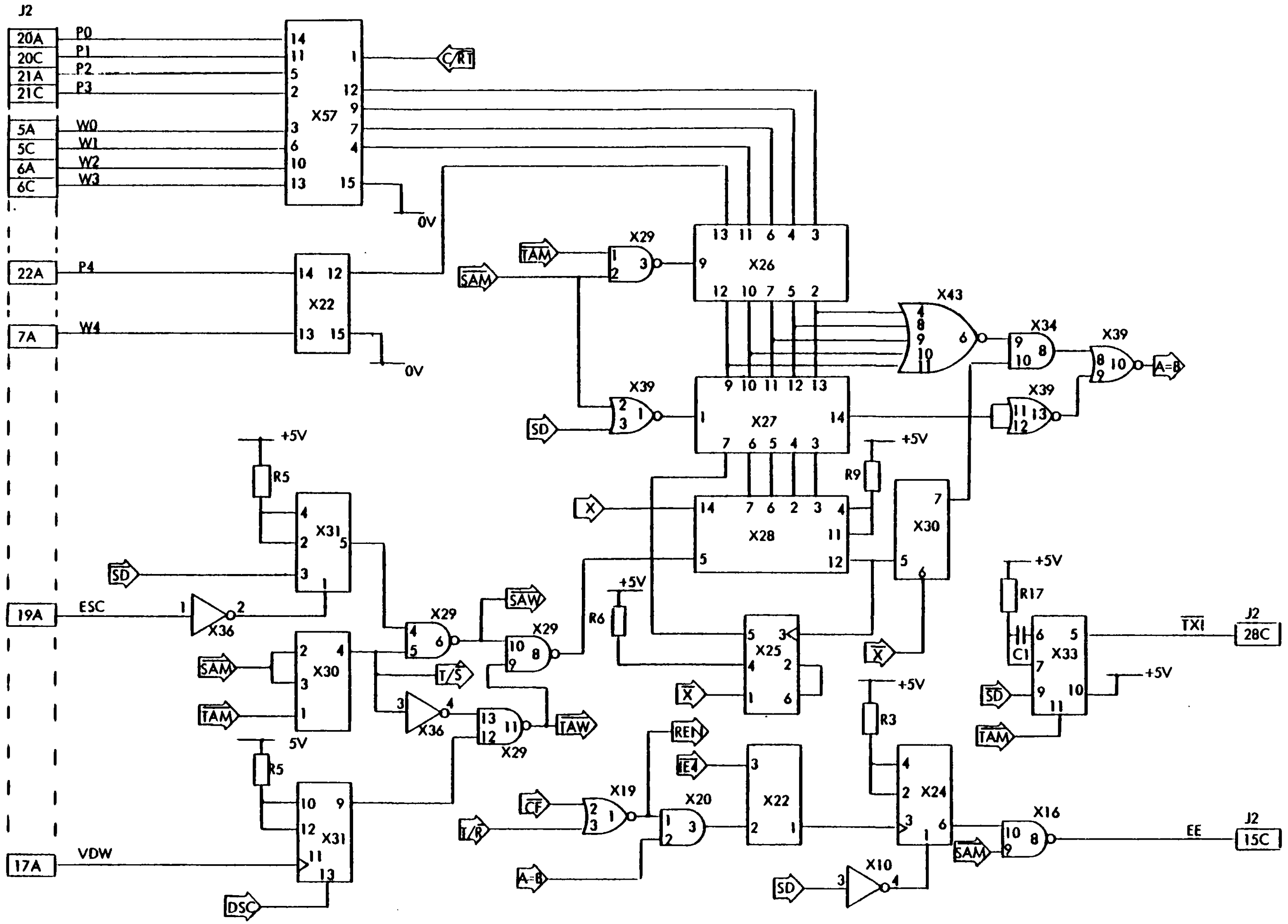


Fig. 10.5.e. 1553B/Microprocessor Interface Board 1.

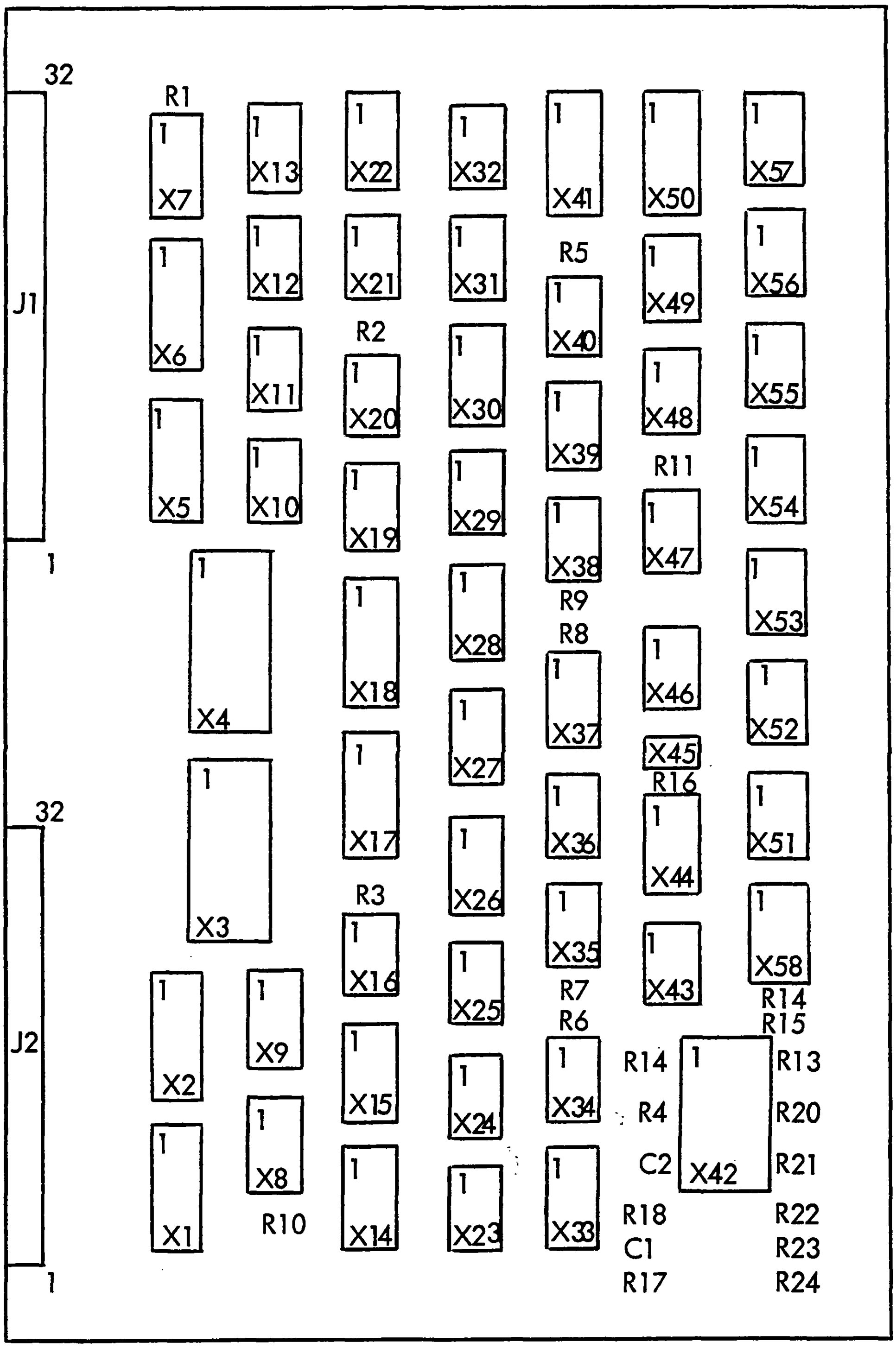


Fig.10.6. Layout of 1553B/Microprocessor Interface Board 1.

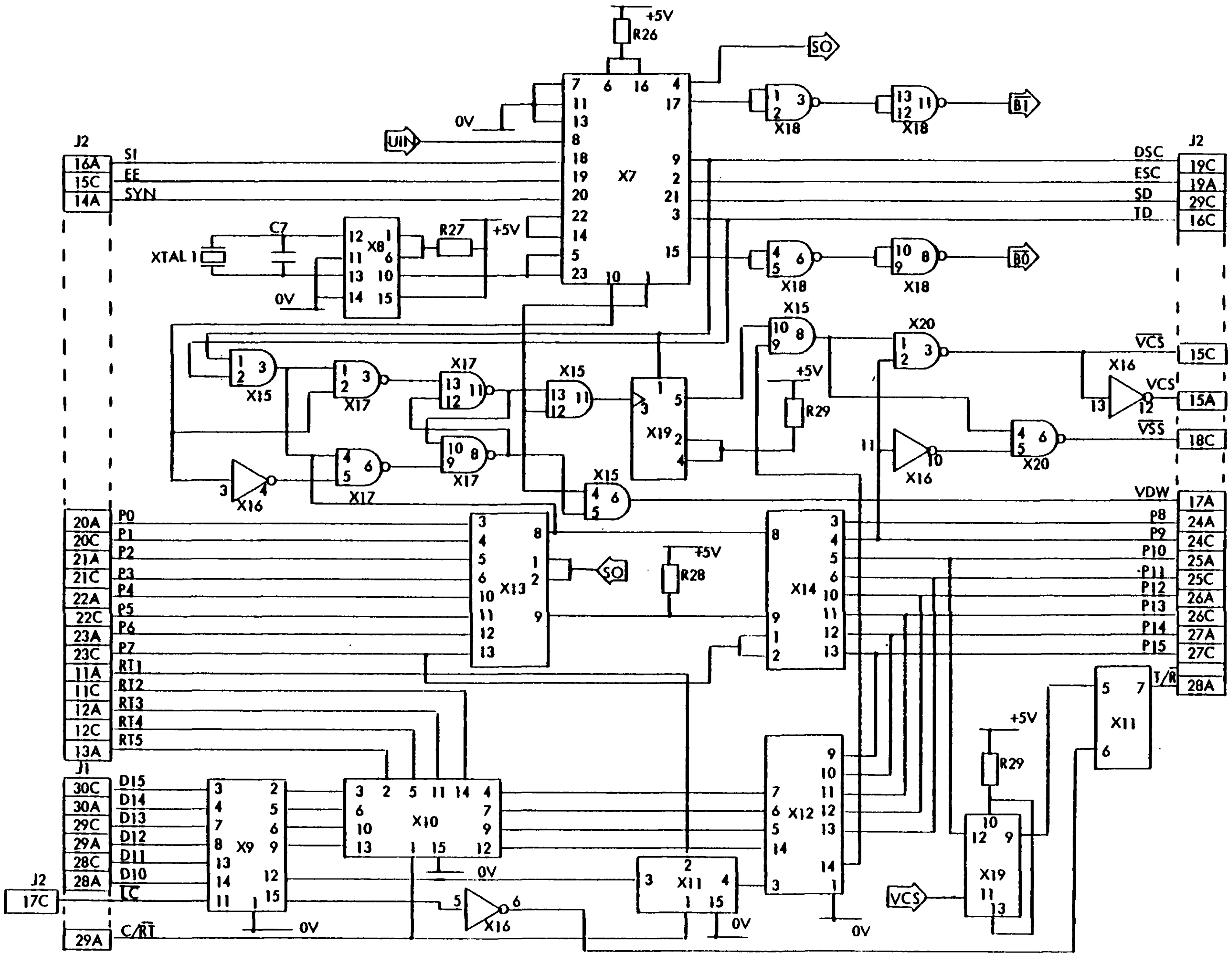


Fig. 10.7.a. 1553B/Microprocessor Interface Board 2.

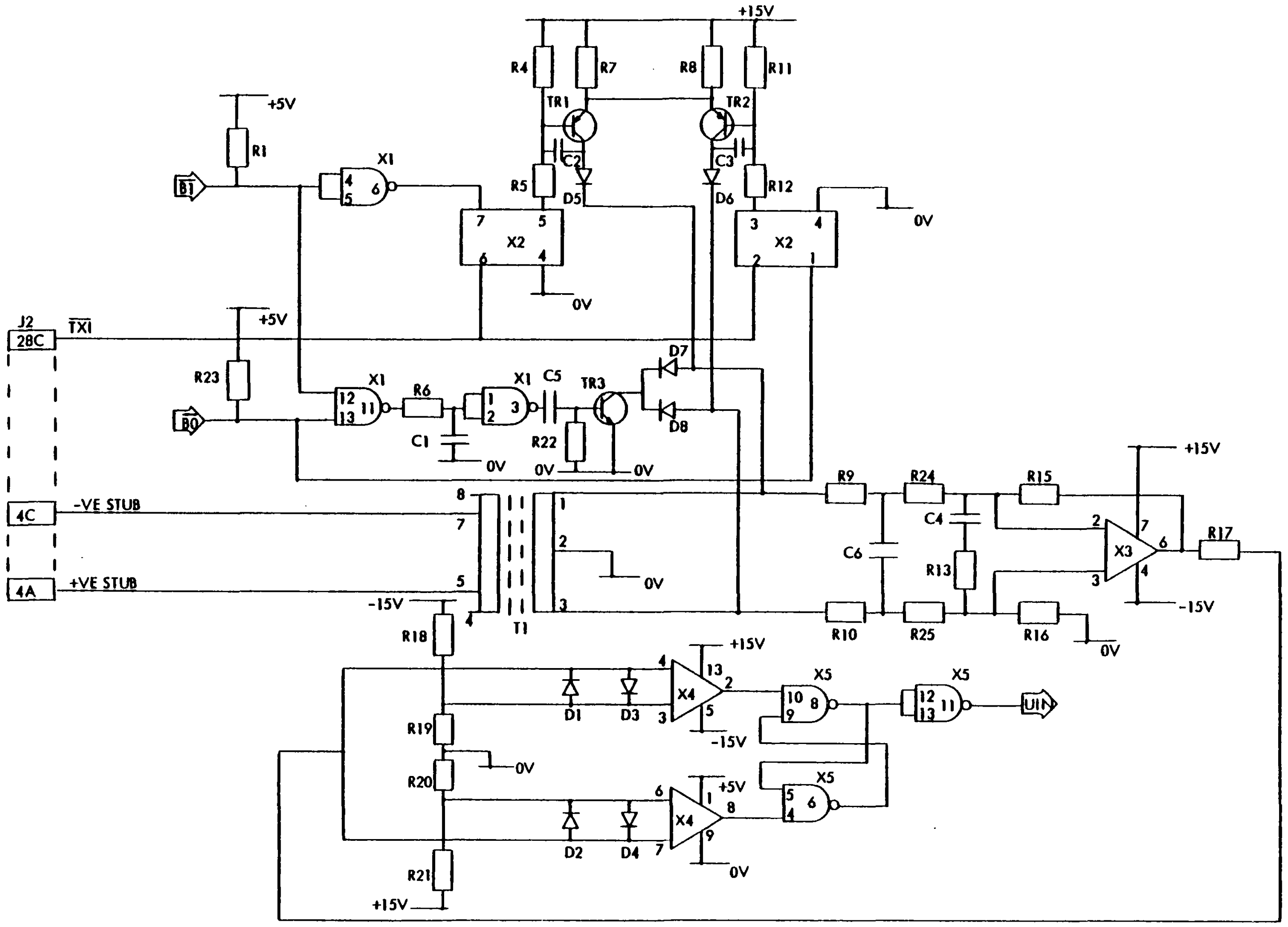


Fig. 10.7.b. 1553B/Microprocessor Interface Board 2.

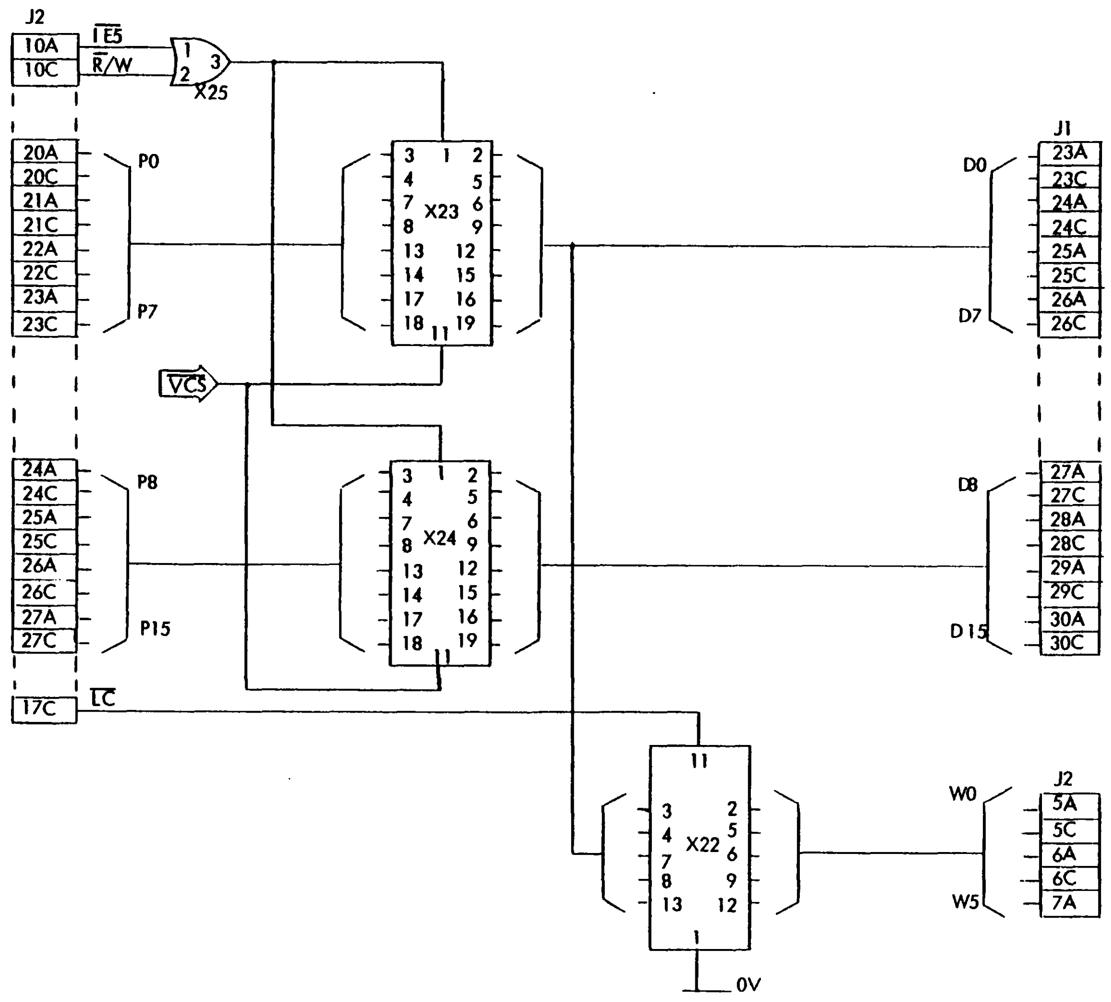


Fig. 10.7. c. 1553B/Microprocessor Inferface Board 2.

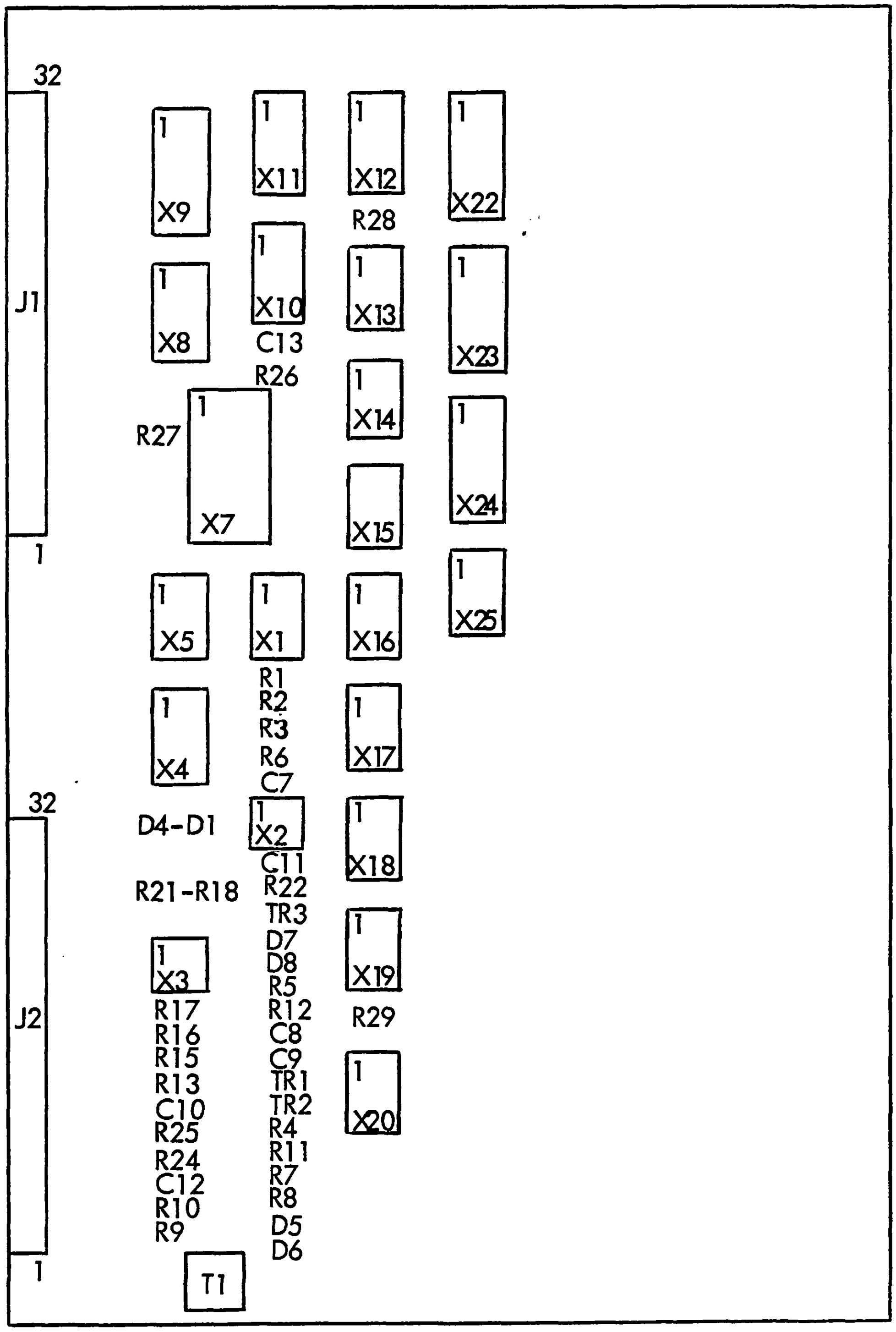


Fig.10.8. Layout of 1553B/Microprocessor Interface Board 2.

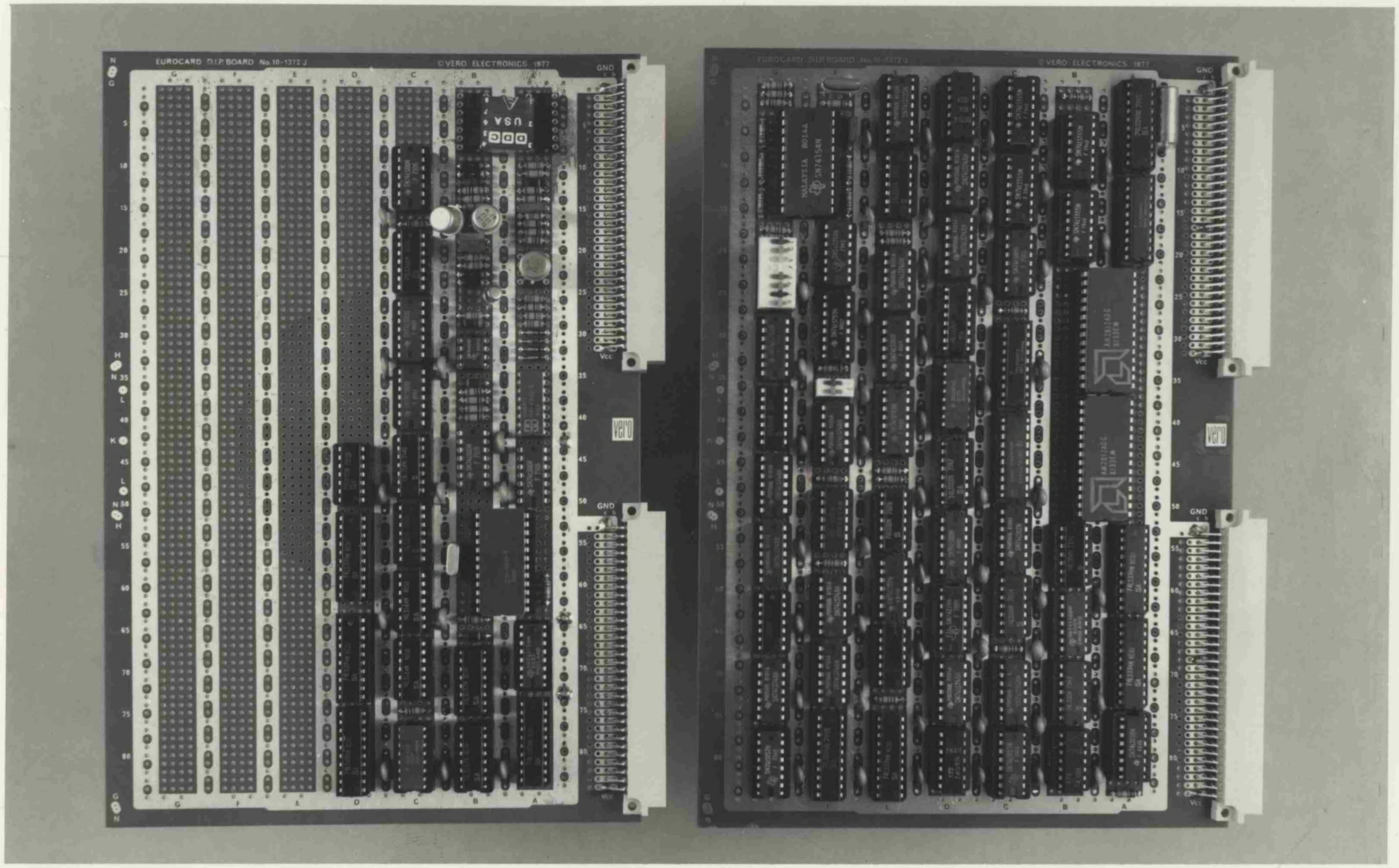


Fig. 10.9.   1553B/Microprocessor Interface Boards.

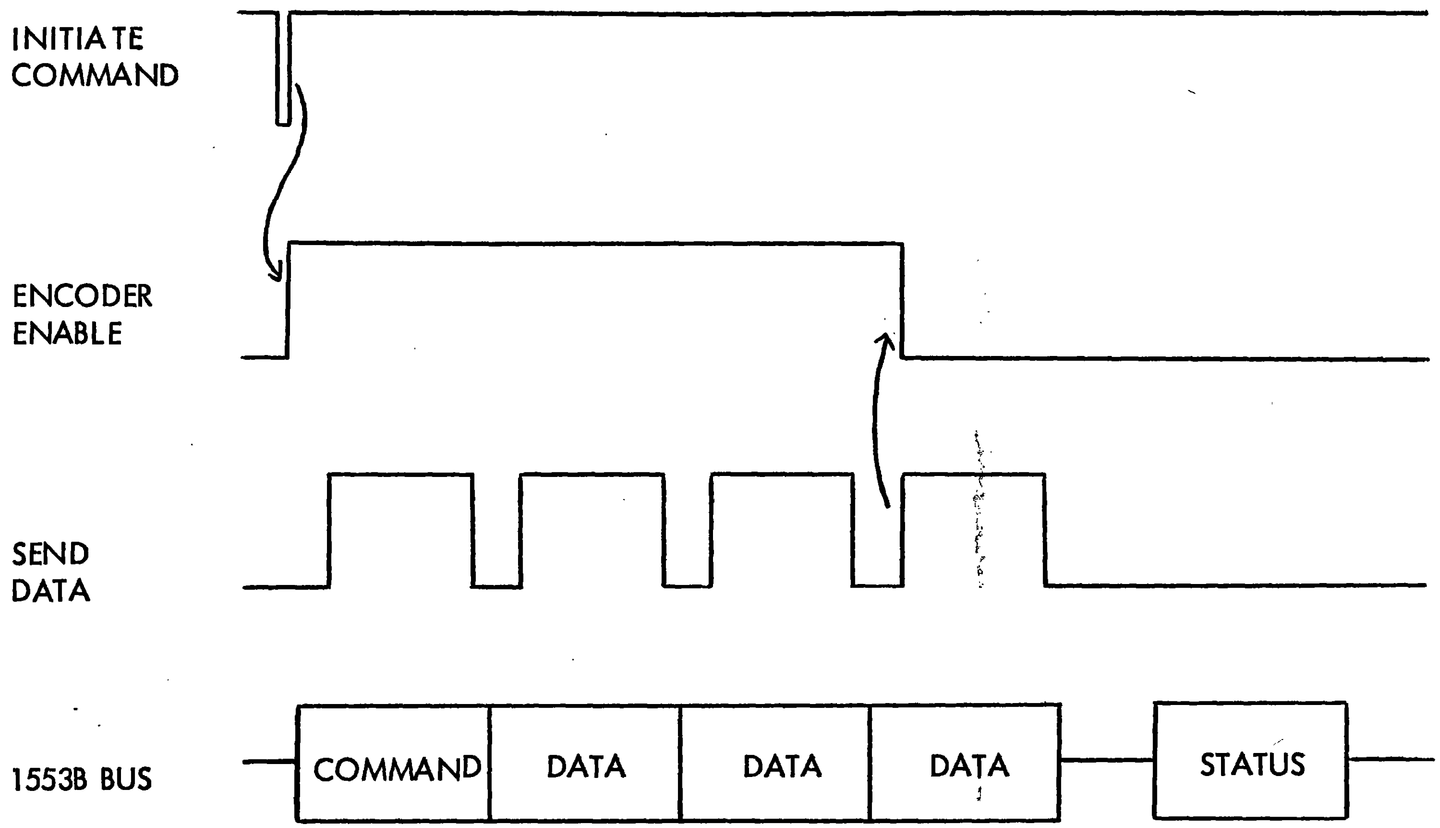


Fig. 10.10. Message from Bus Controller: Hardware Operation.

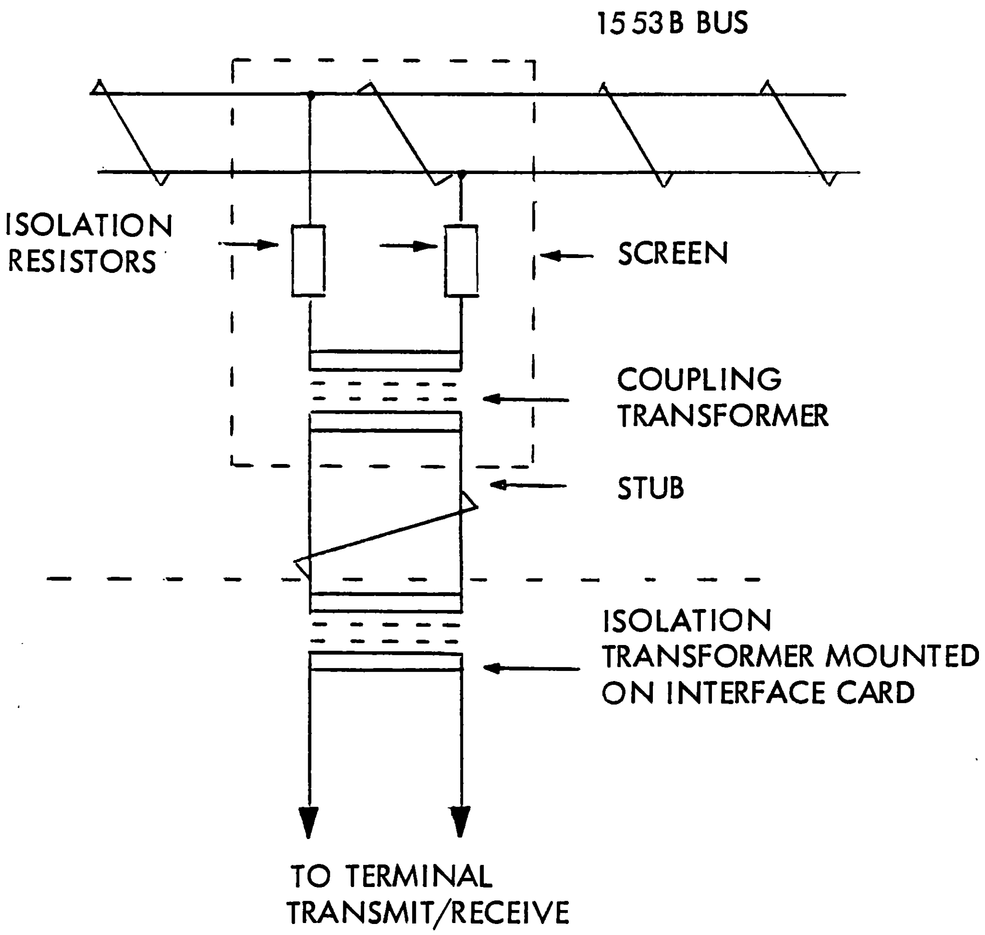

1553B BUS
ISOLATION RESISTORS
SCREEN
COUPLING TRANSFORMER
STUB
ISOLATION TRANSFORMER MOUNTED ON INTERFACE CARD
TO TERMINAL TRANSMIT/RECEIVE

Fig.10.11. Connection of Terminal to 1553B Bus.

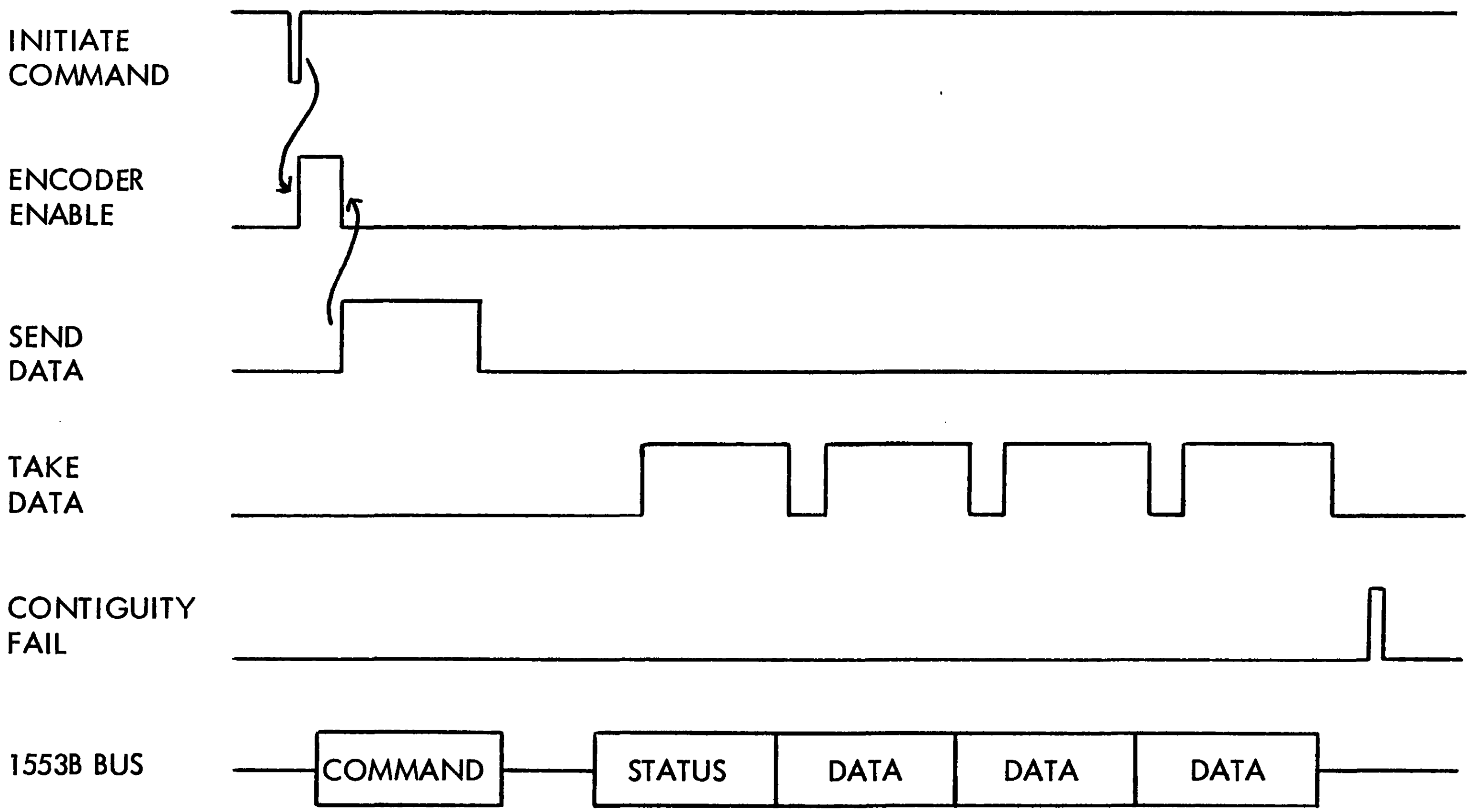


Fig. 10.12. Message to Bus Controller: Hardware Operation.

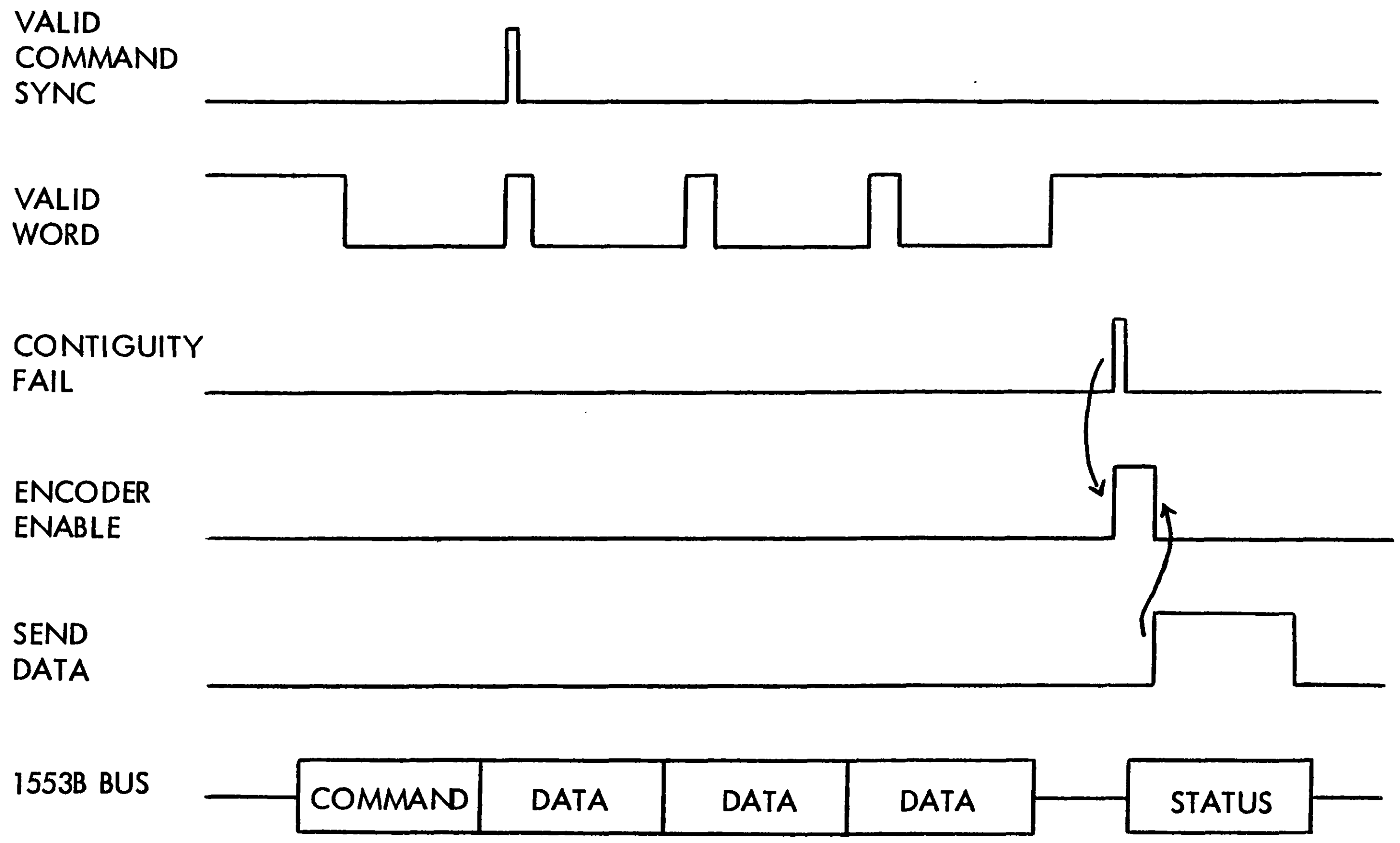


Fig. 10.13. Message to Remote Terminal: Hardware Operation.

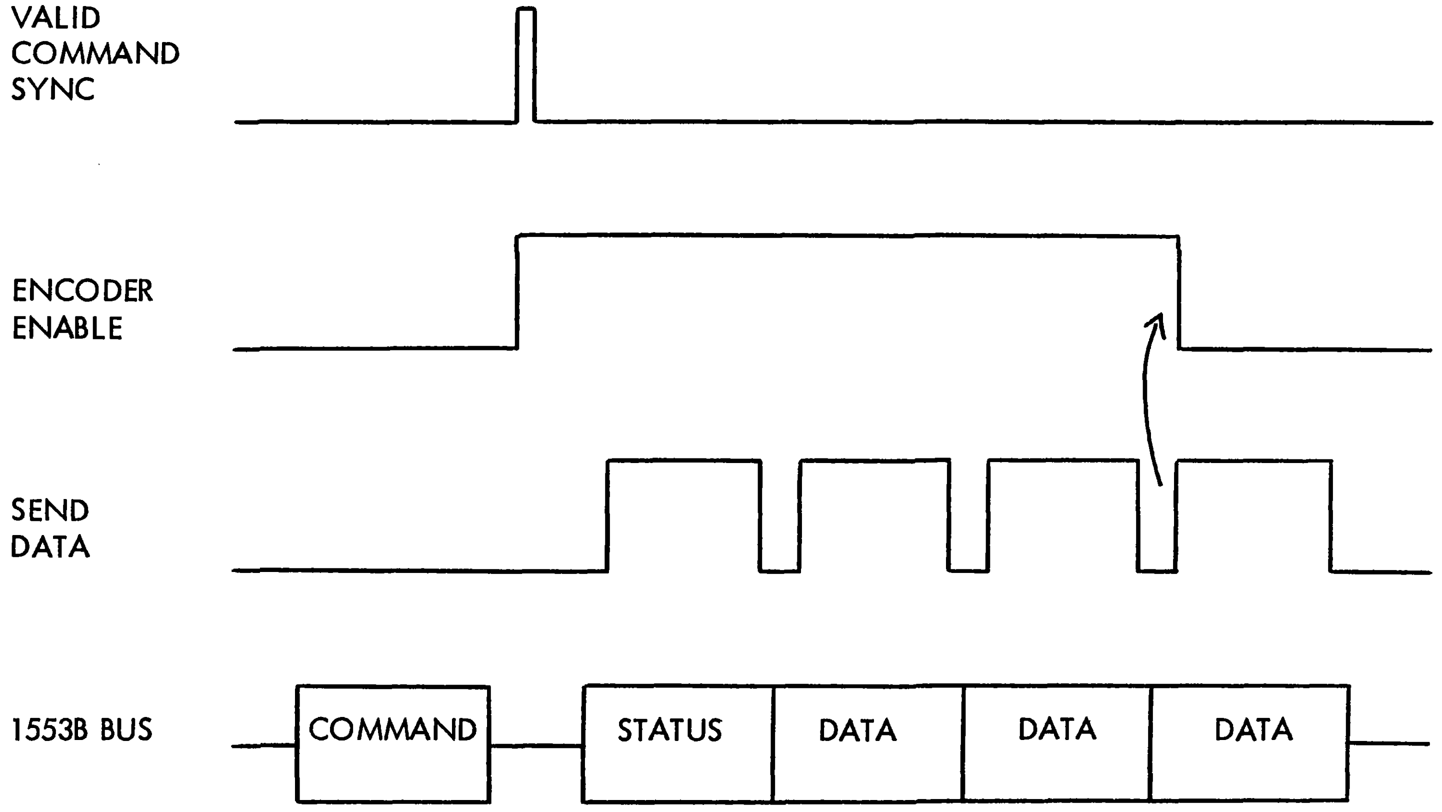


Fig. 10. 14. Message from Remote Terminal: Hardware Operation.

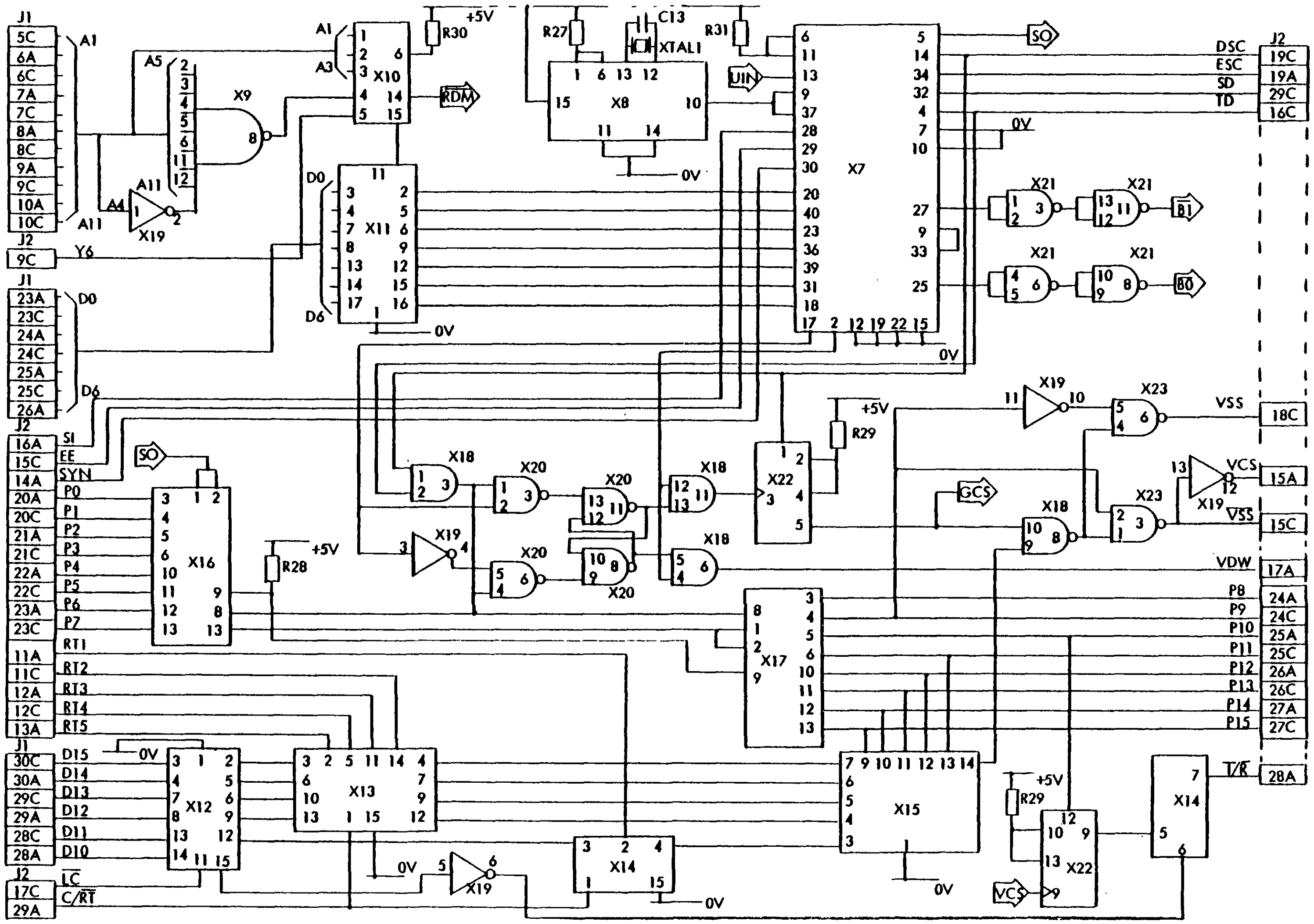


Fig. 10.15.a. 1553B Protocol Fault Injection Board.

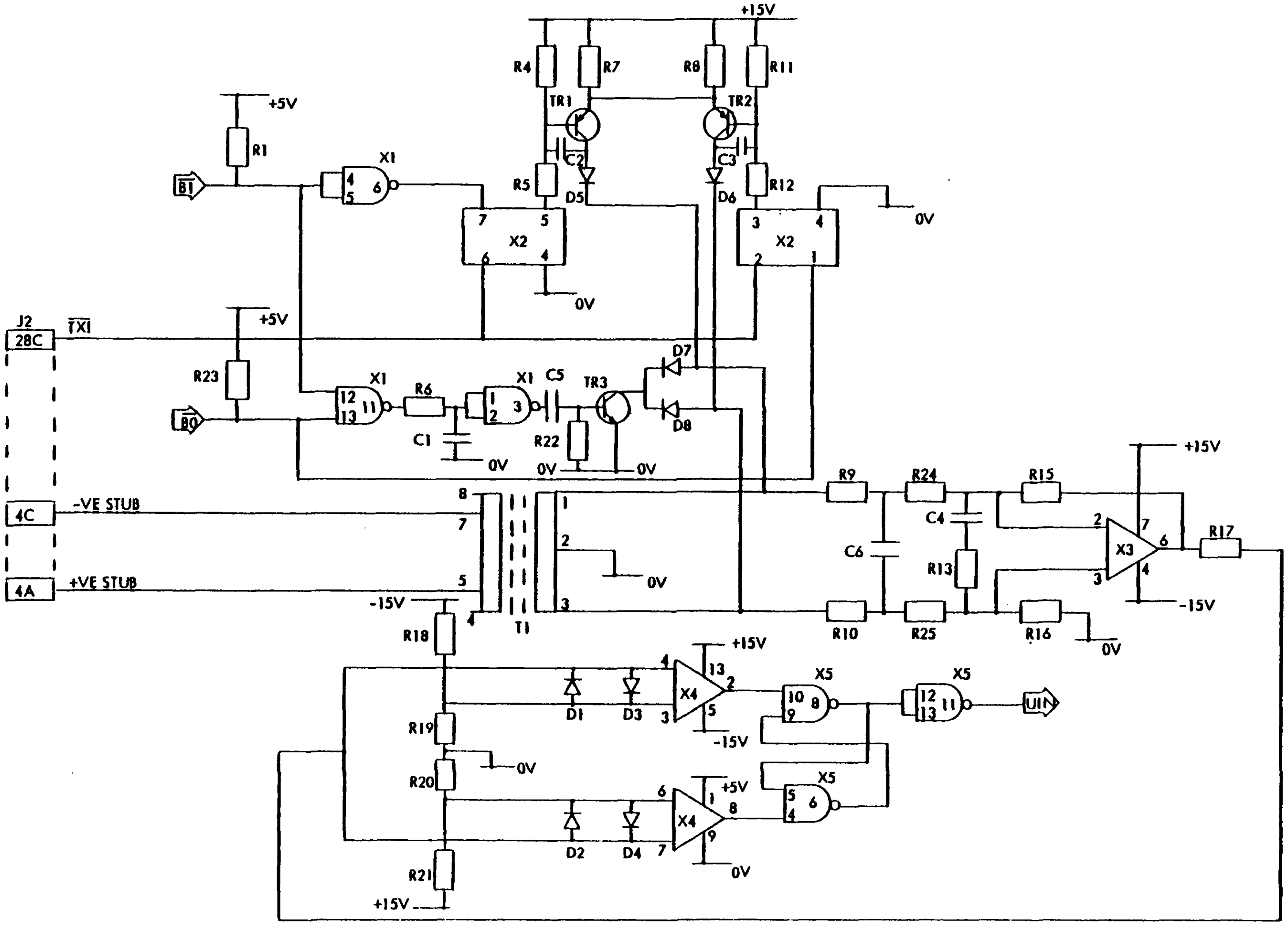


Fig. 10. 15. b. 1553B Protocol Fault Injection Board.

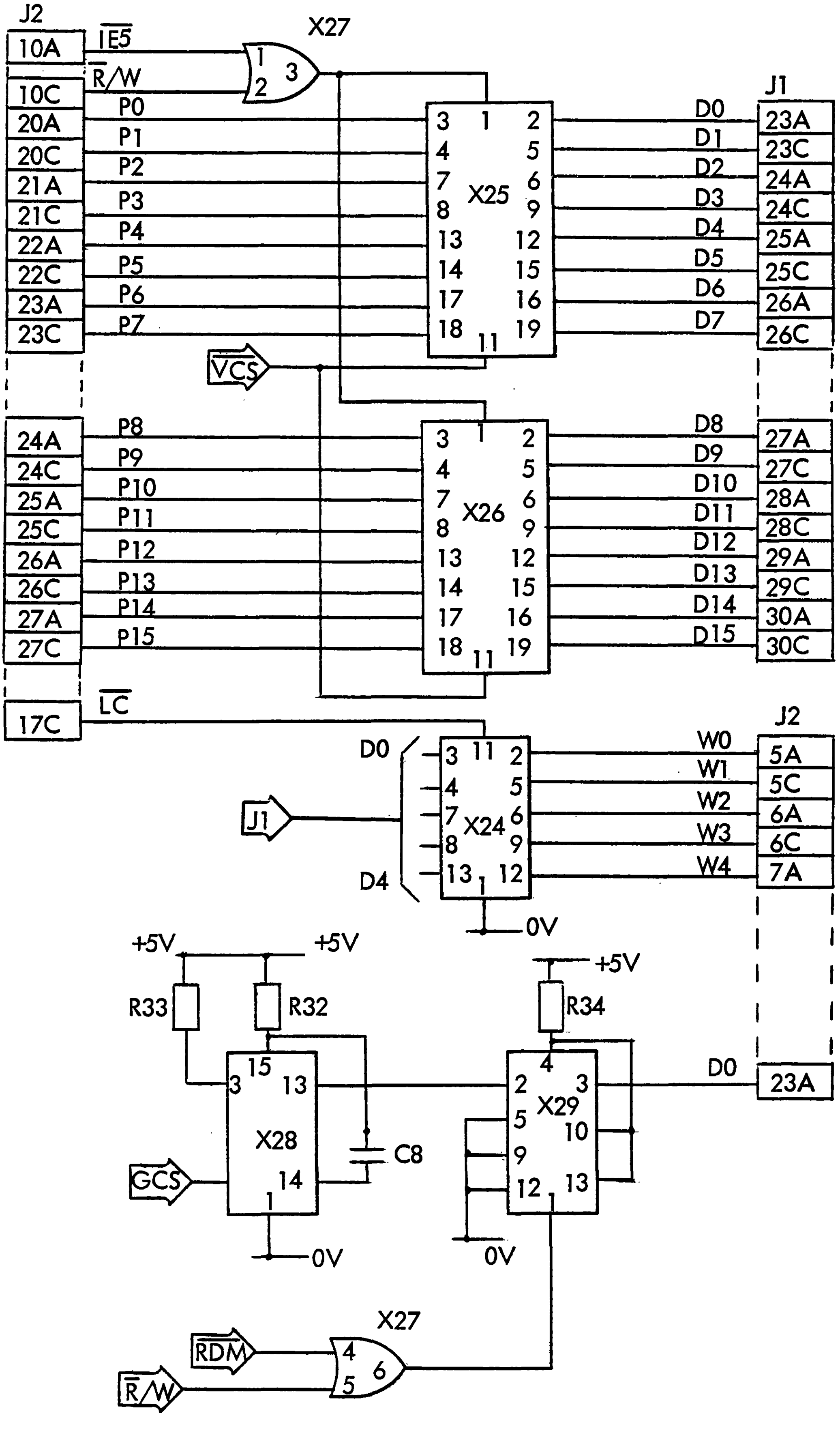


Fig. 10. 15. c. 1553B Protocol Fault Injection Board.

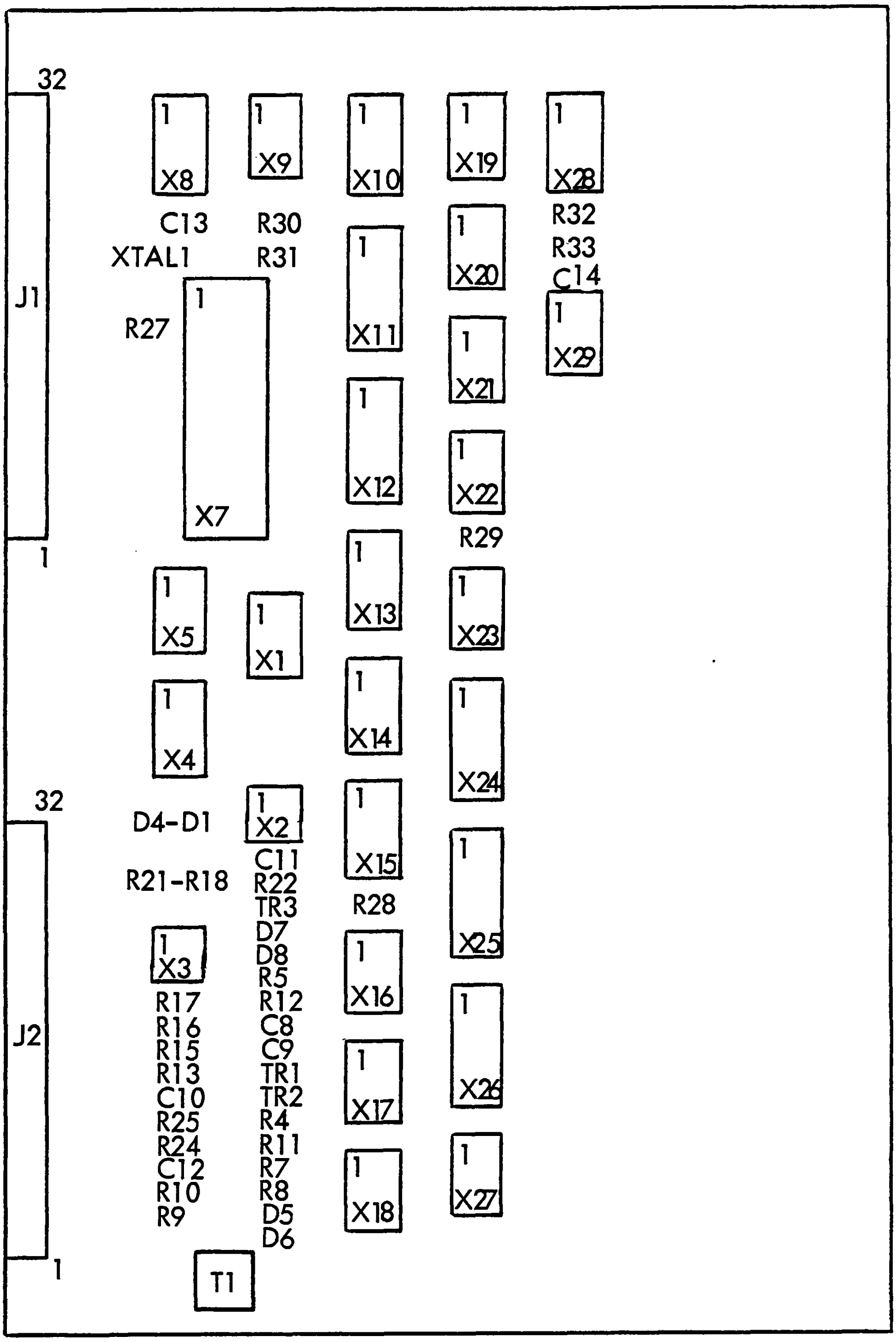


Fig.10.16. Layout of 1553B Protocol Fault Injection Board.

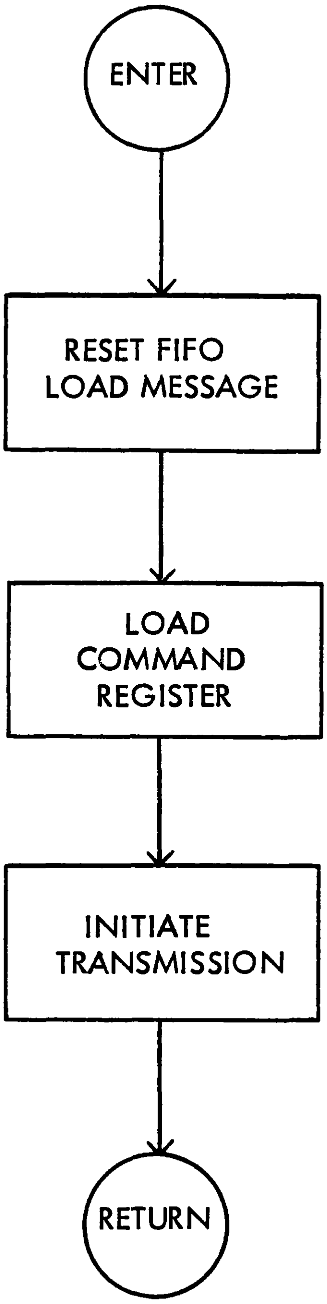


Fig.10.17. Message from Bus Controller: Software Operation.

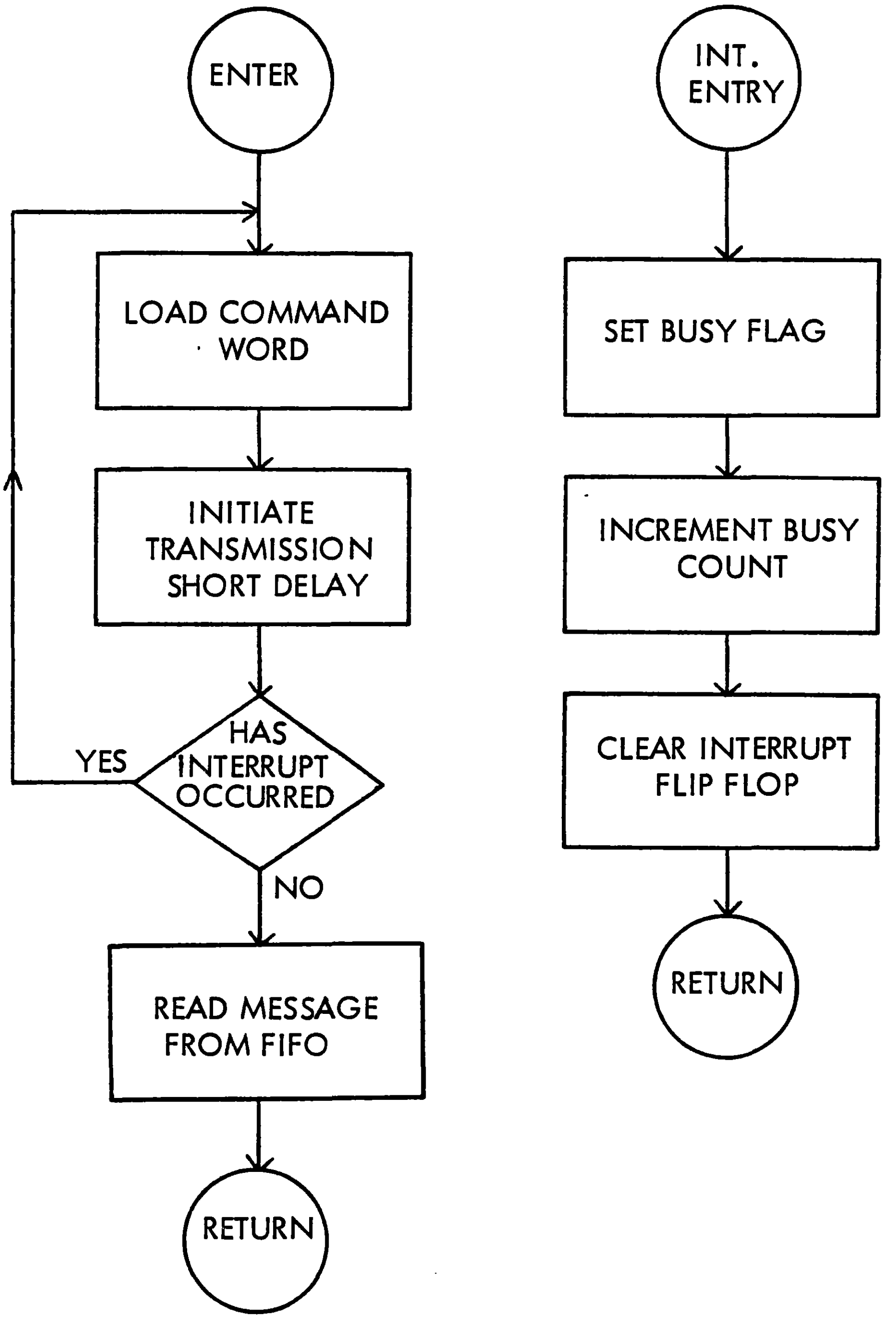


Fig. 10.18. Message to Bus Controller: Software Operation.

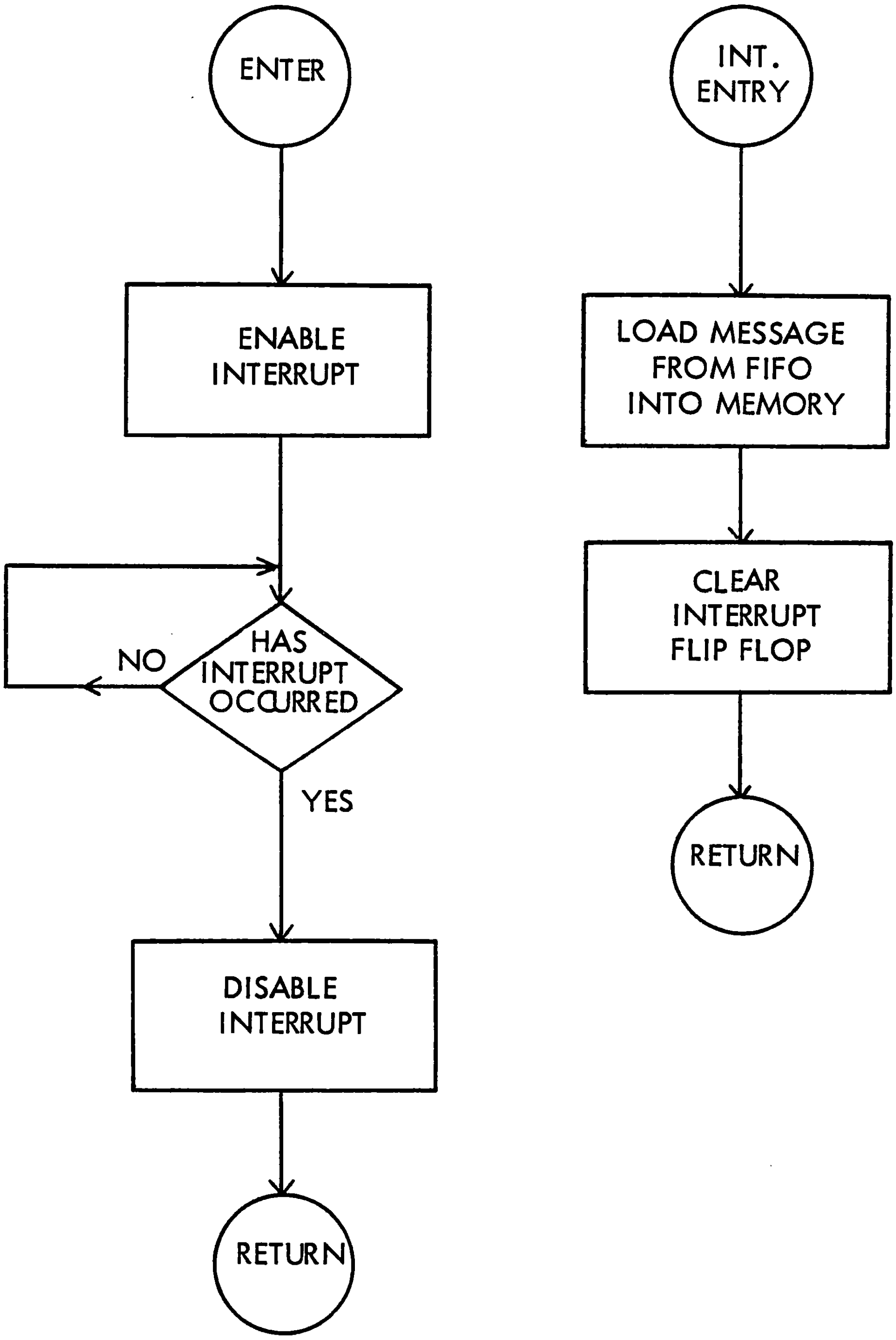


Fig. 10.19. Message to Remote Terminal: Software Operation.

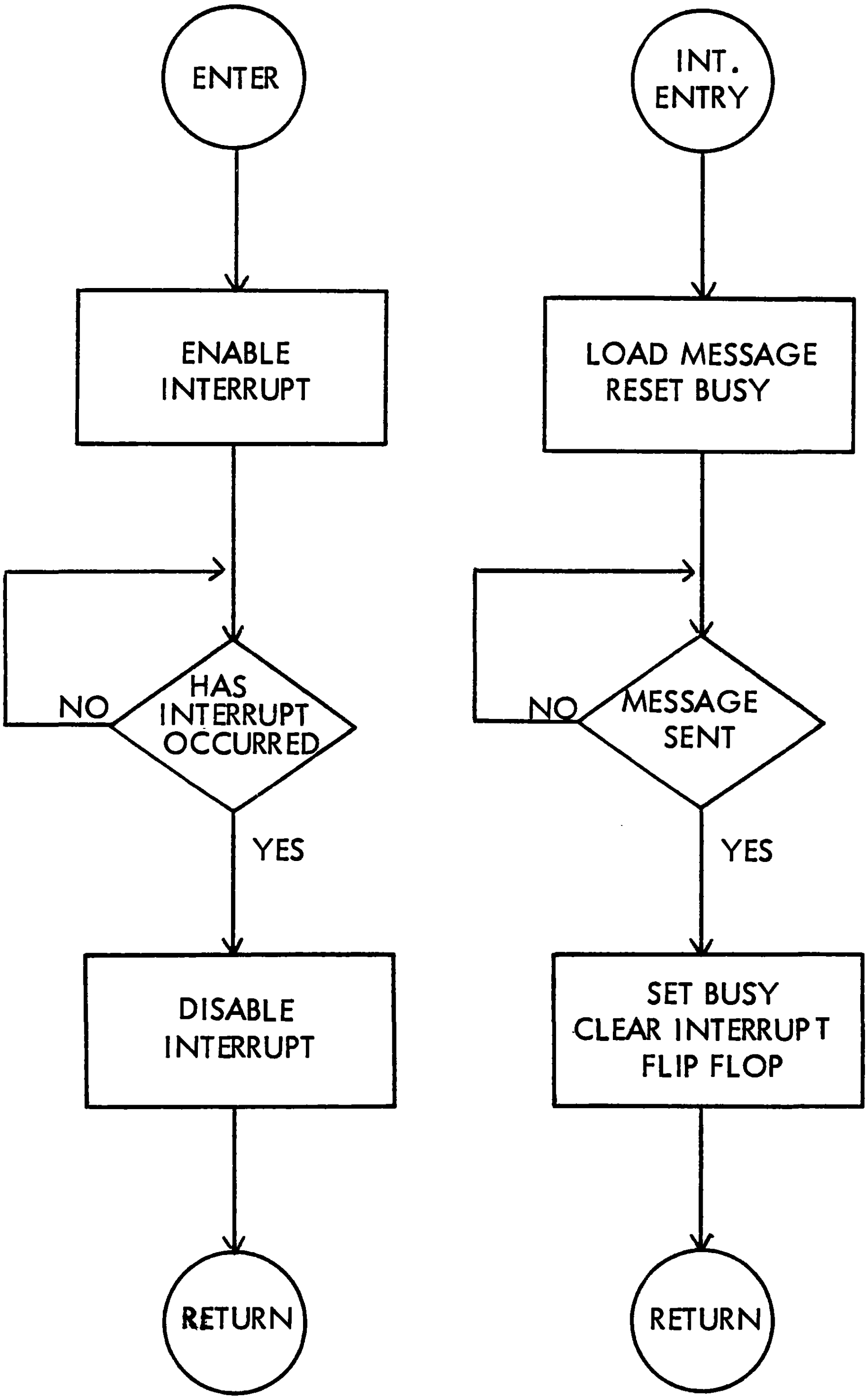


Fig.10.20. Message from Remote Terminal: Software Operation.

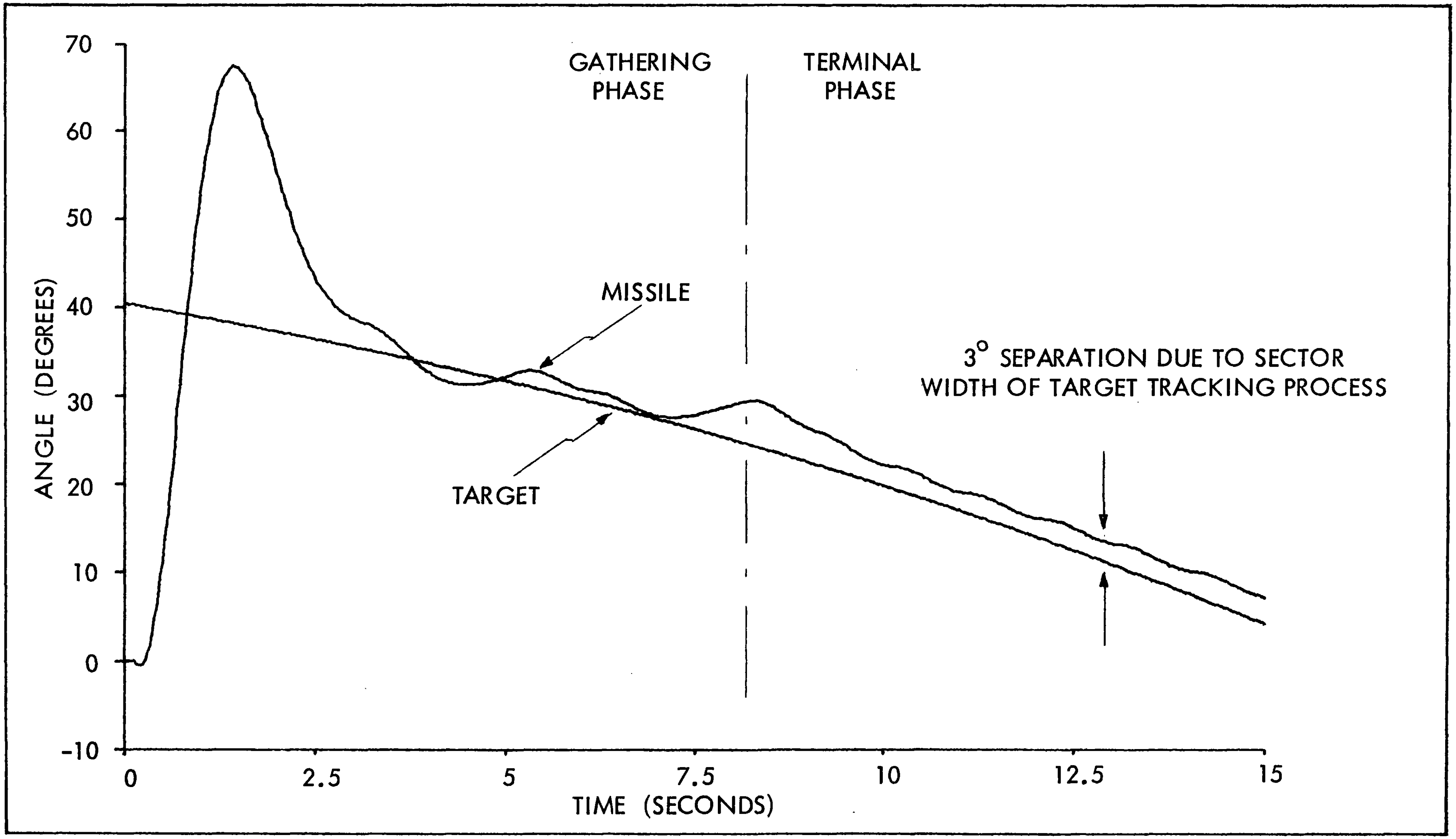


Fig.11.1. Missile Angle Plot.

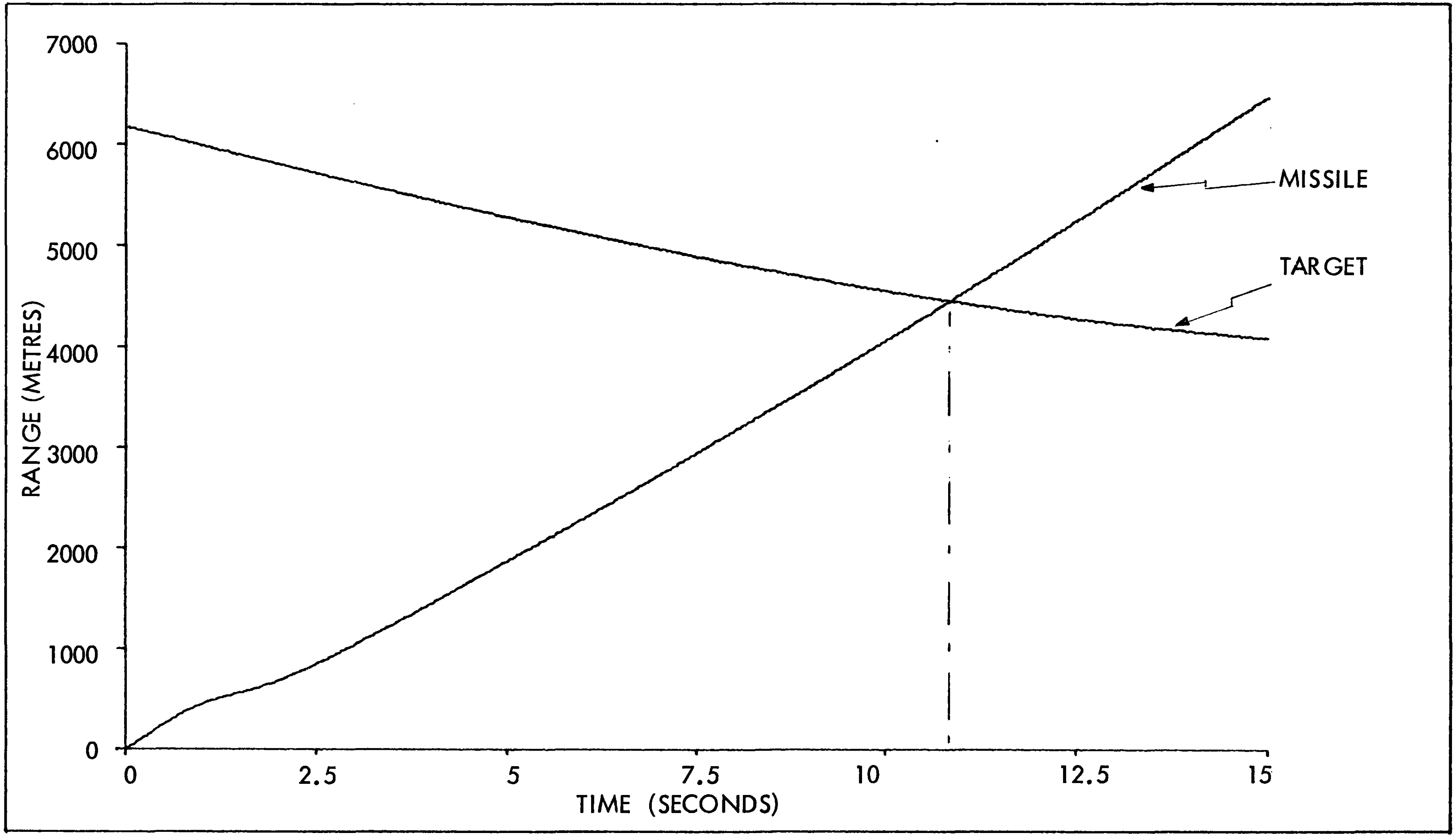


Fig.11.2. Missile Range Plot.

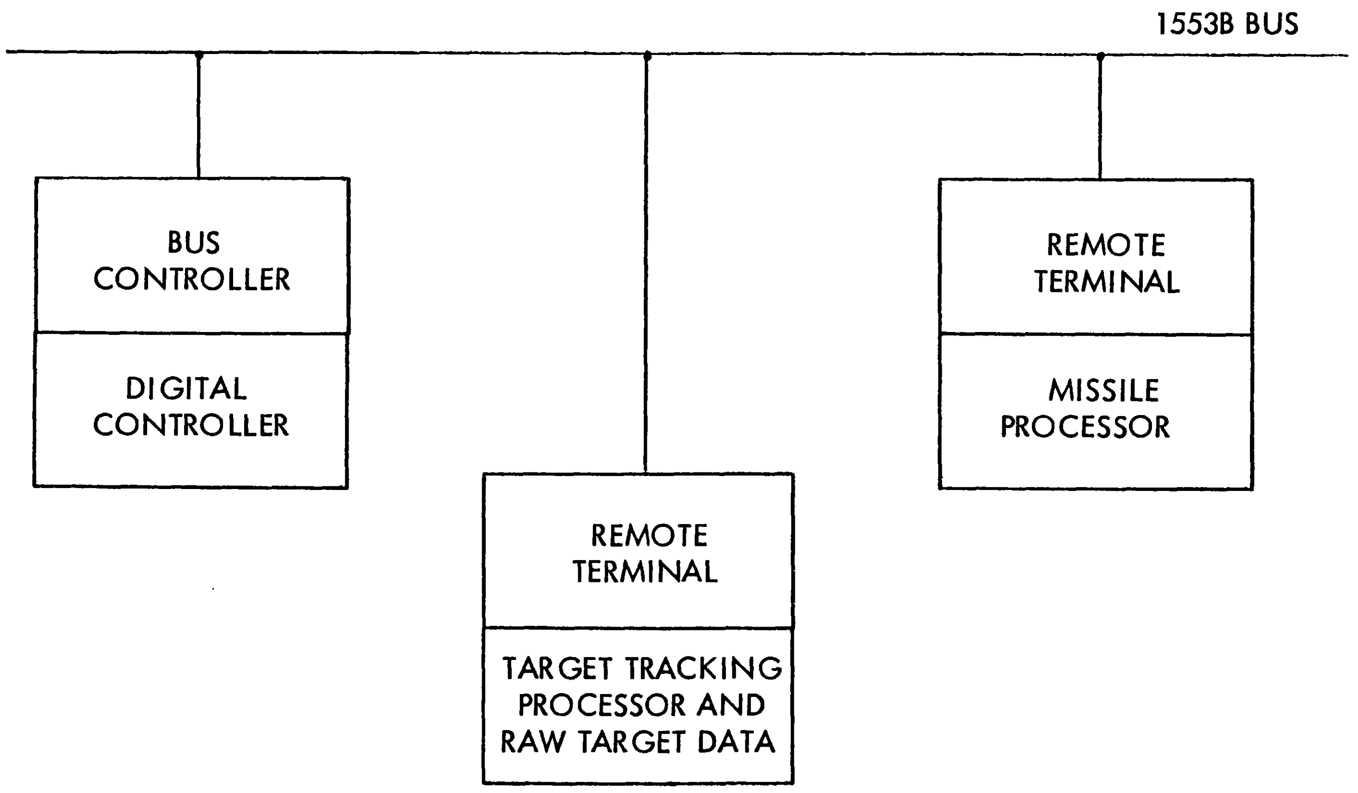


Fig. 11.3. System Configuration for Baseline Results.

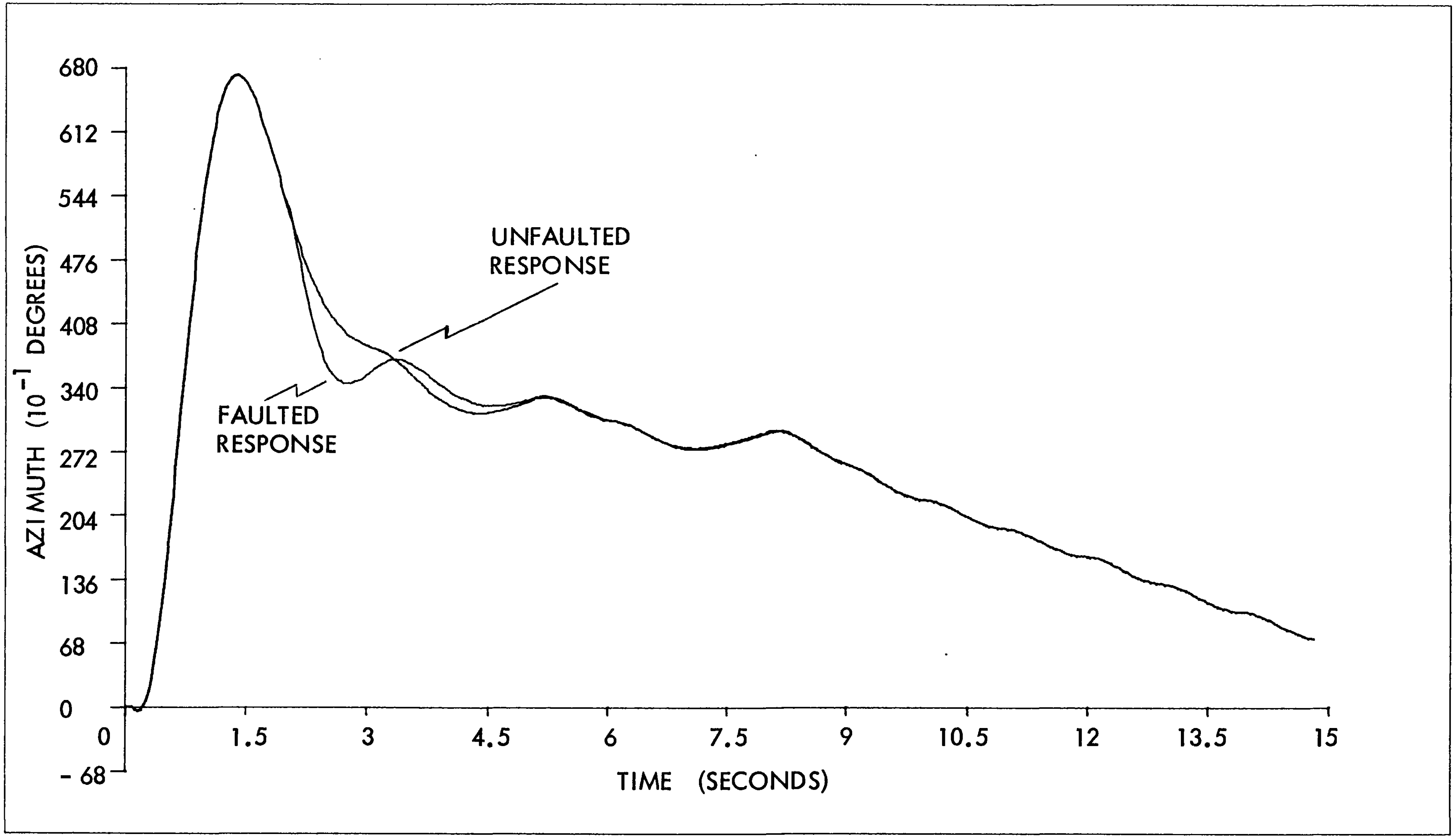


Fig. 11.4. Data Corruption Type Fault in Gathering Phase (2 seconds)

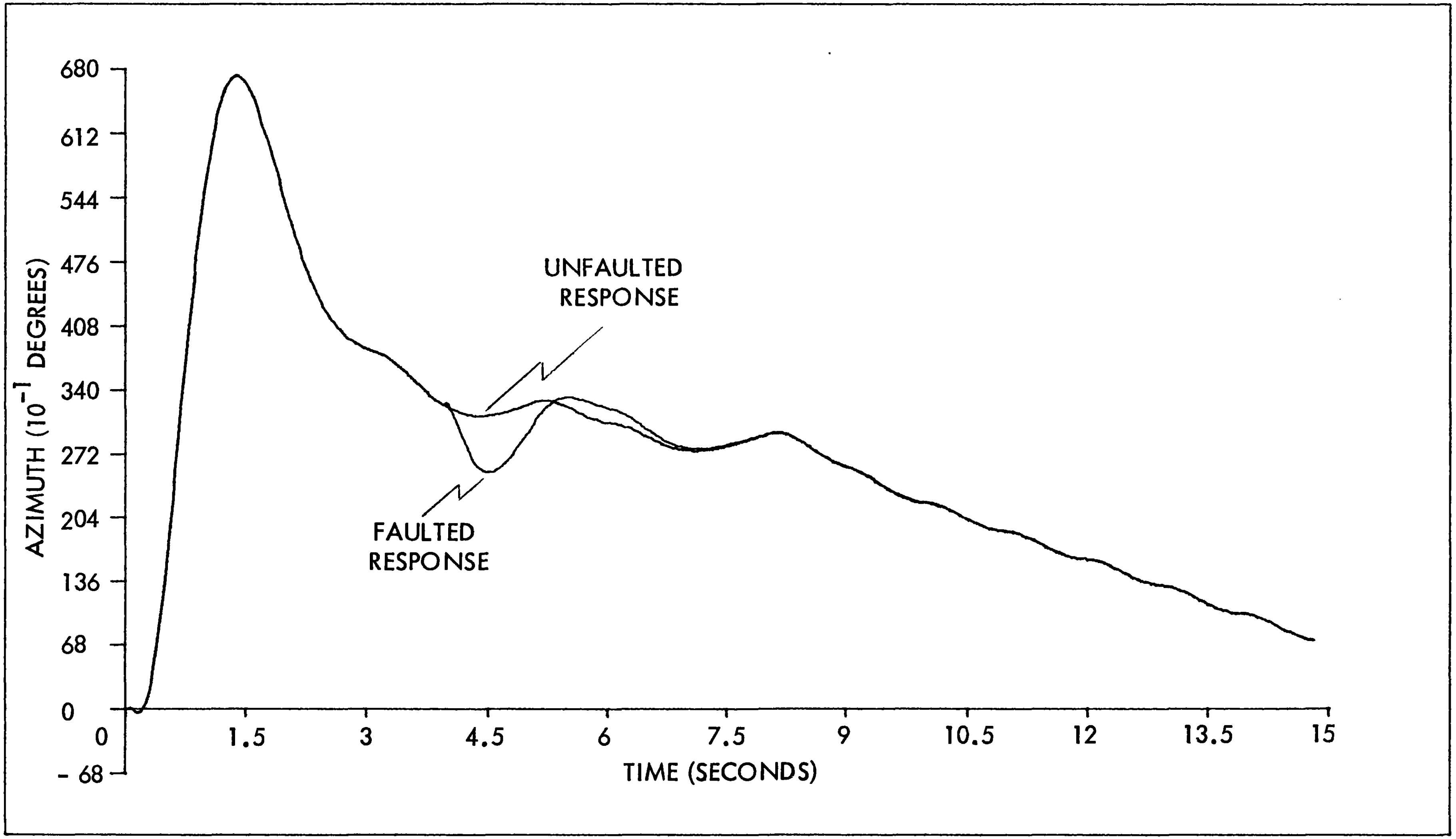


Fig. 11.5. Data Corruption Type Fault in Gathering Phase (4 seconds).

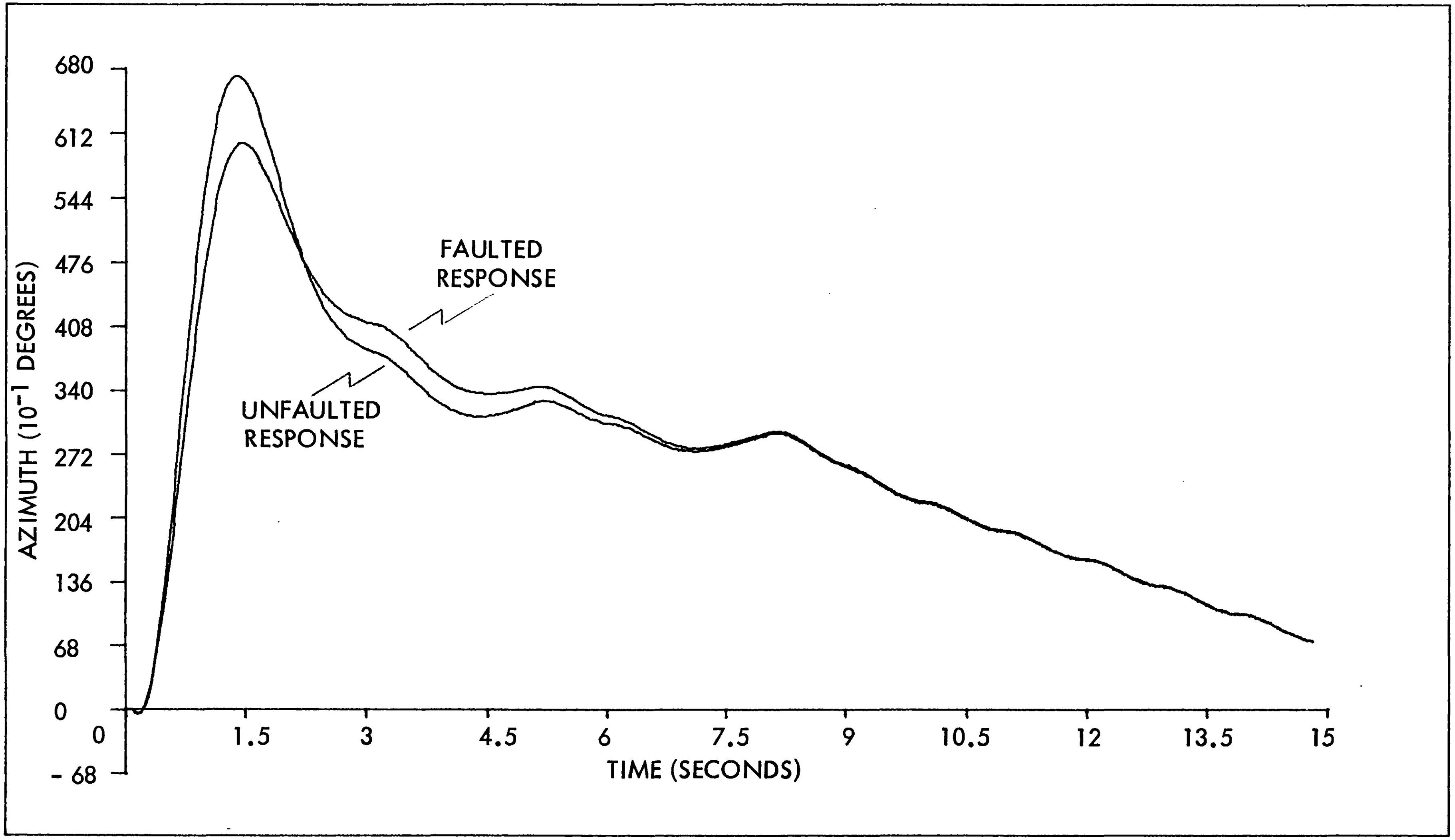


Fig.11.6. Data Corruption Type Fault in Gathering Phase ($\frac{1}{4}$ second).

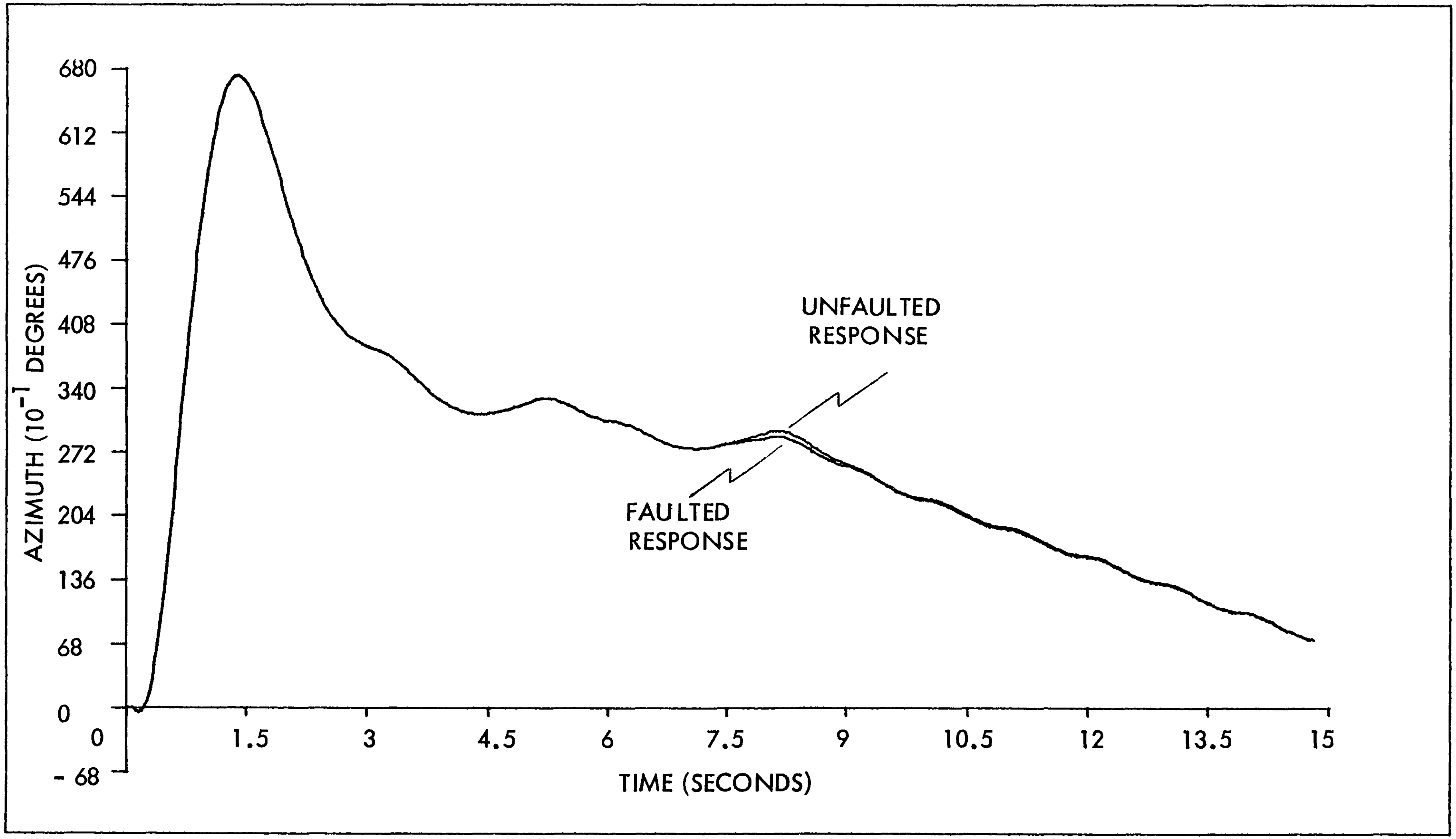


Fig. 11.7. Data Corruption Type Fault in Terminal Phase (7½ seconds).

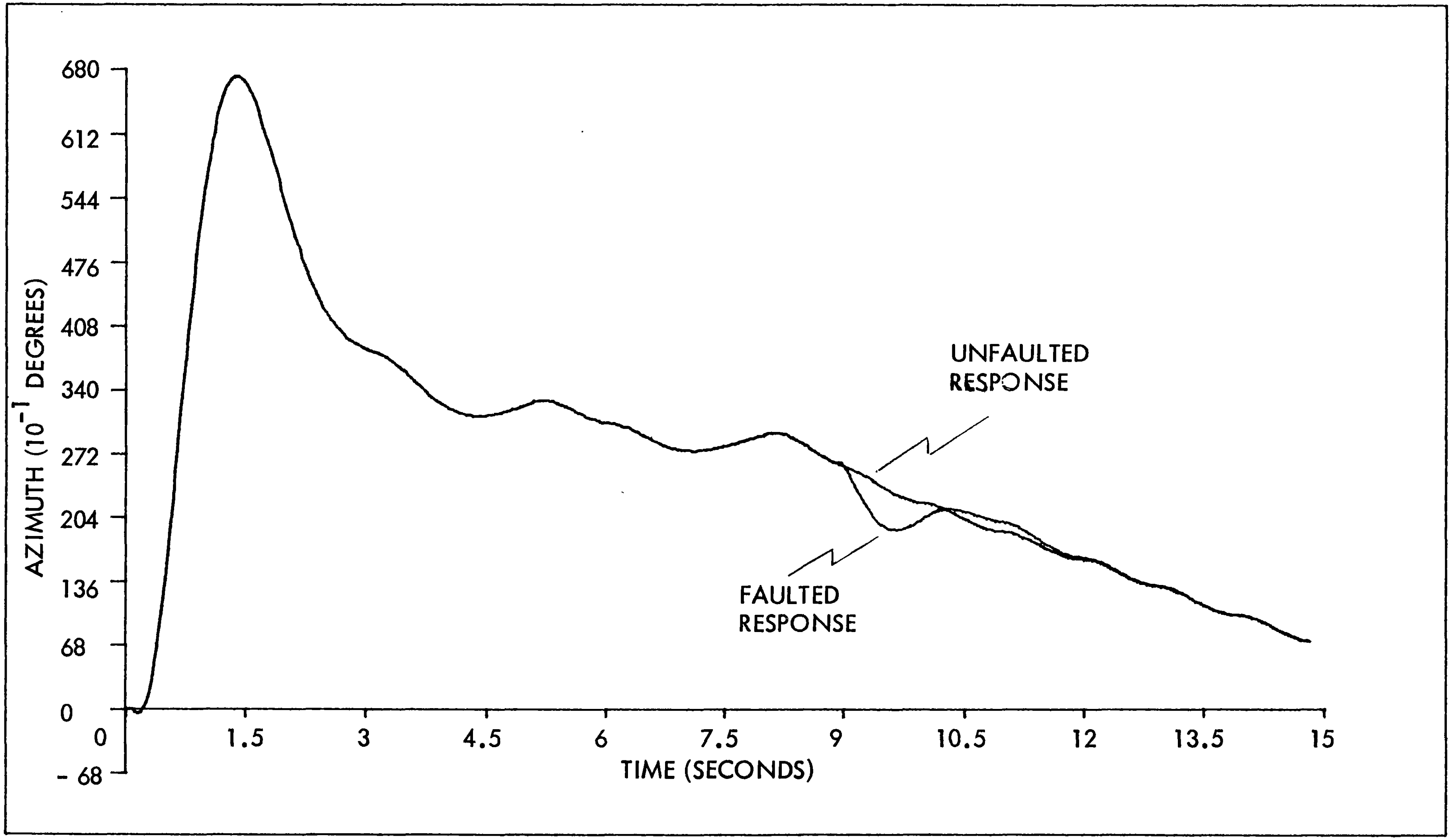


Fig.11.8. Data Corruption Type Fault in Terminal Phase (9 seconds).

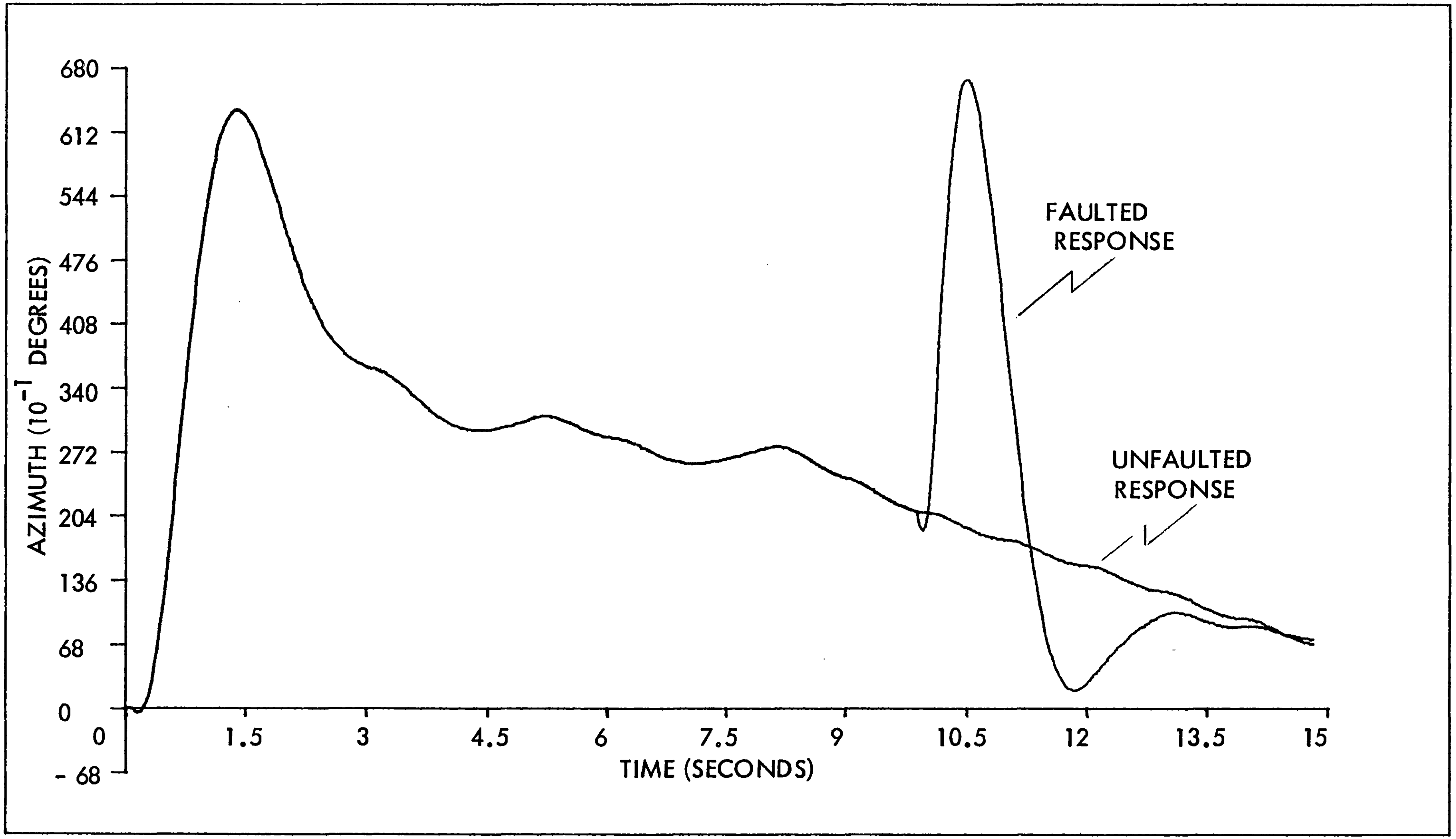


Fig. 11.9. Effect of Overflow in Digital Controller.

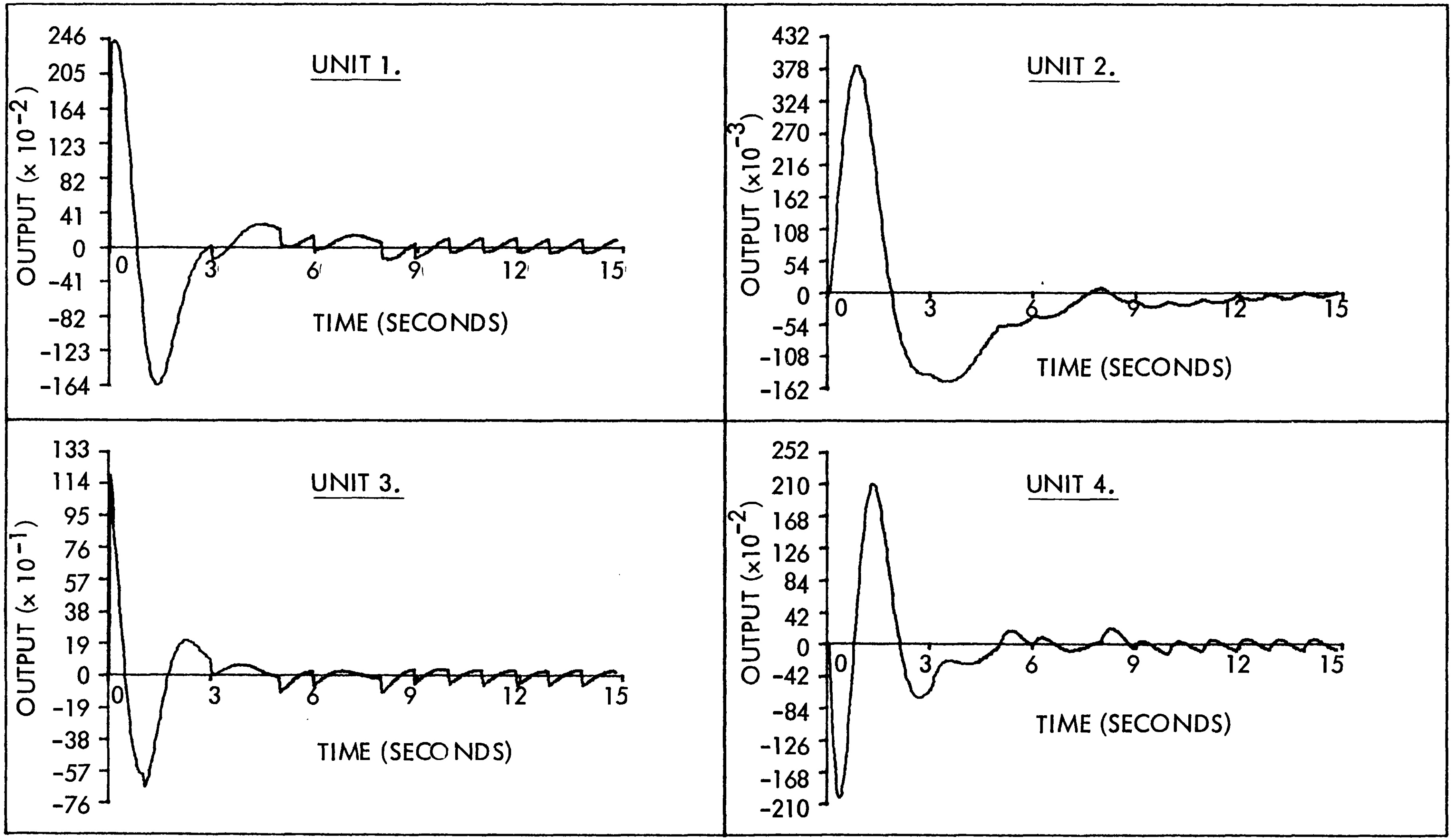


Fig. 11.10. Parallel Units of Digital Controller.

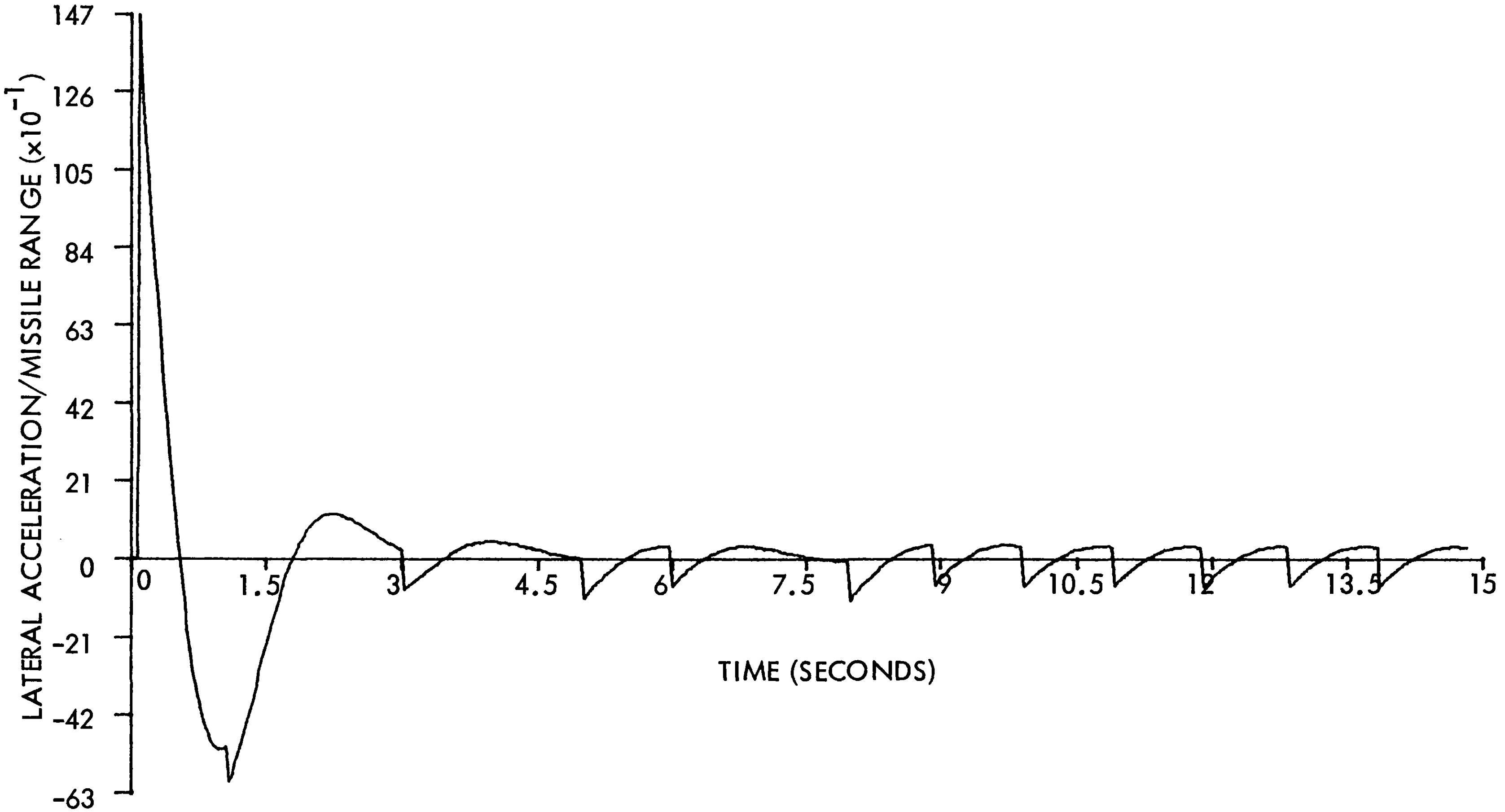


Fig.11.11. Guidance Demand.

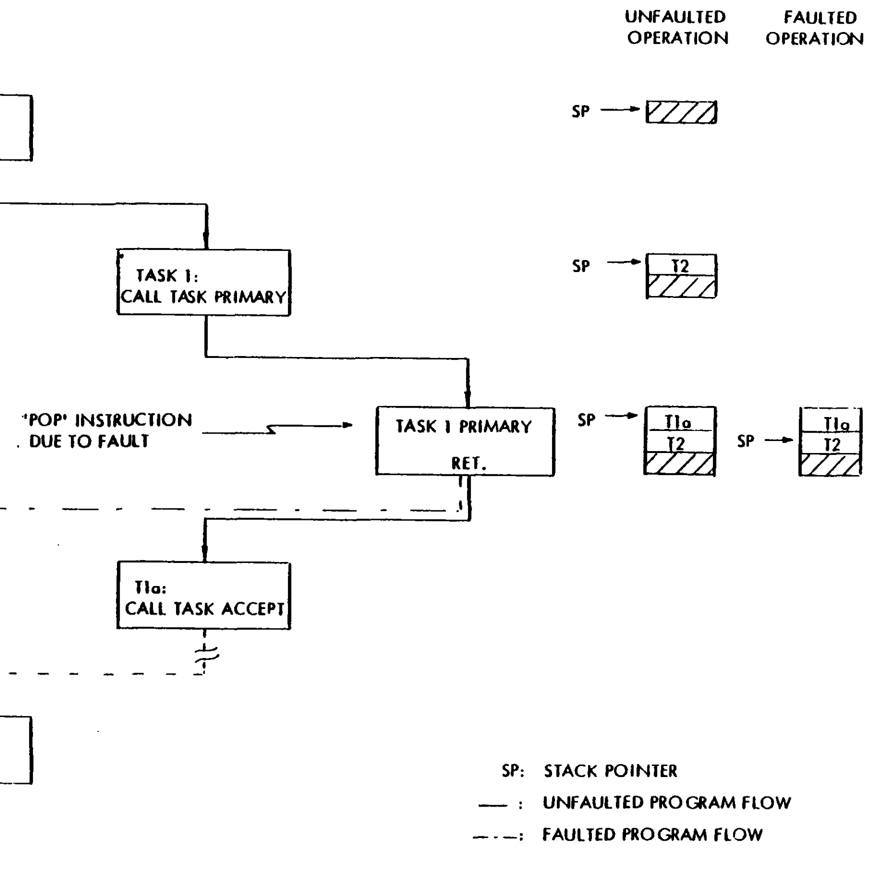


Fig. 11.12. Omission of Acceptance Test due to a Fault.

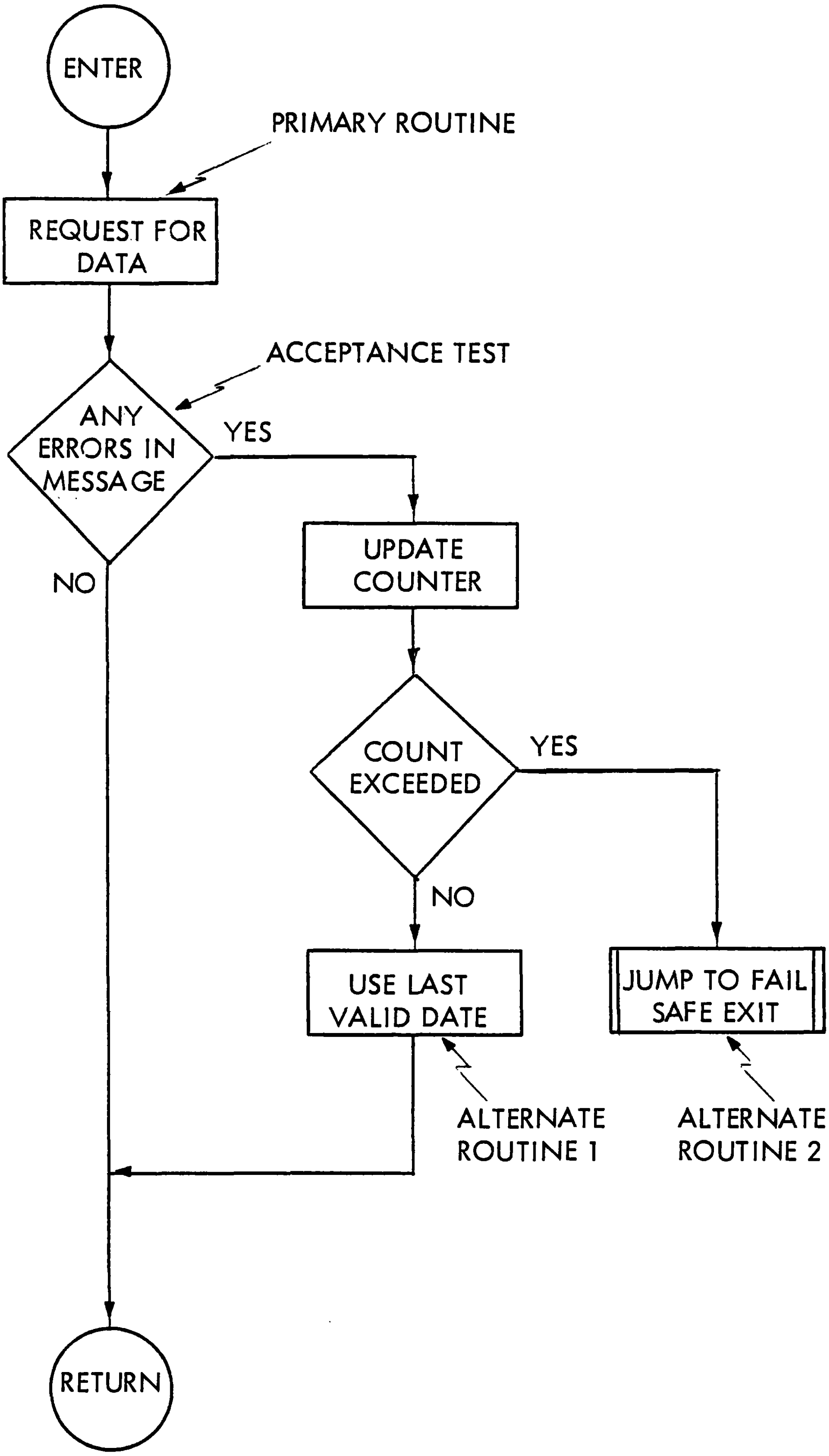


Fig.11.13. System Recovery.

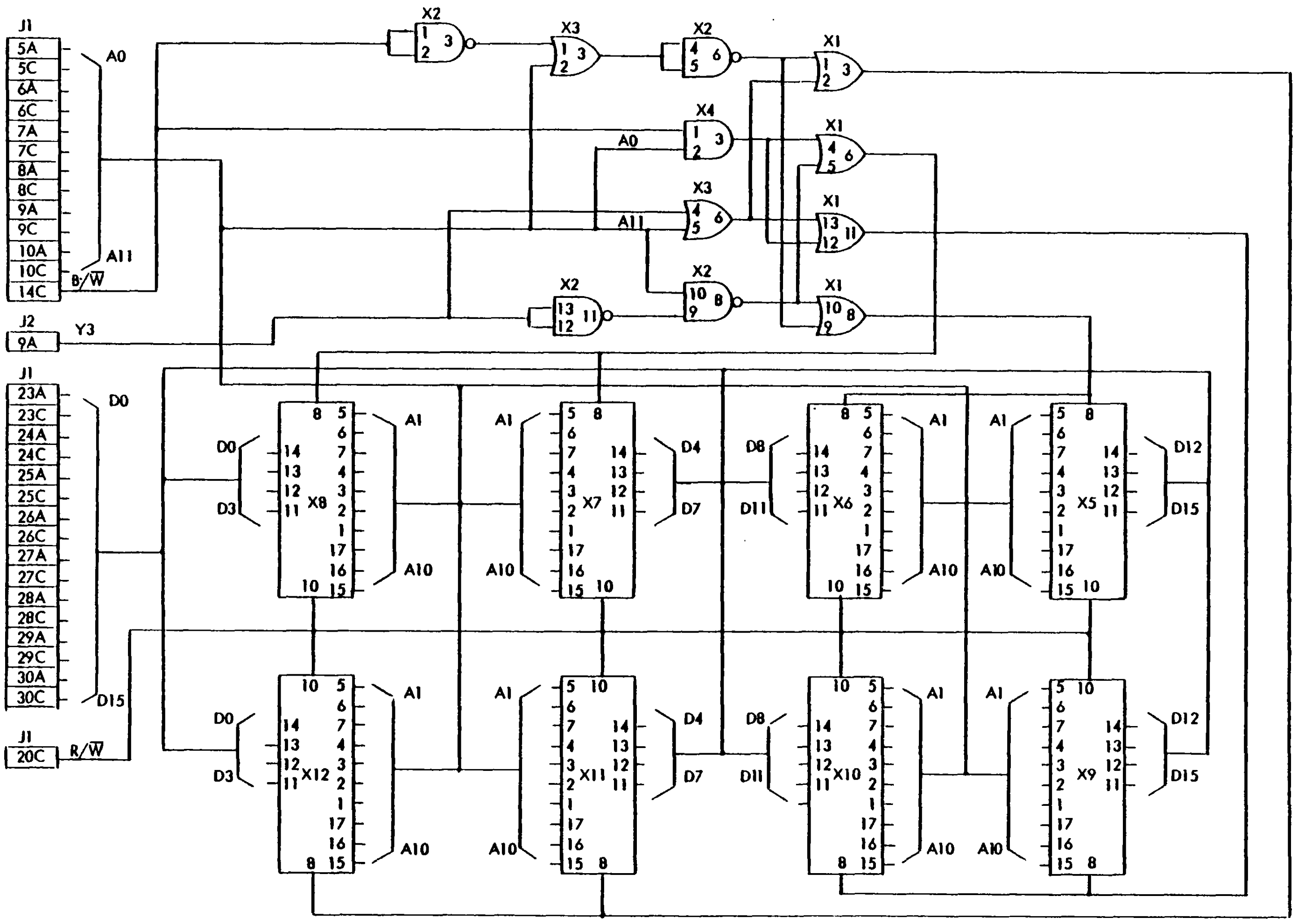


Fig. 12.1. 4K Memory Board.

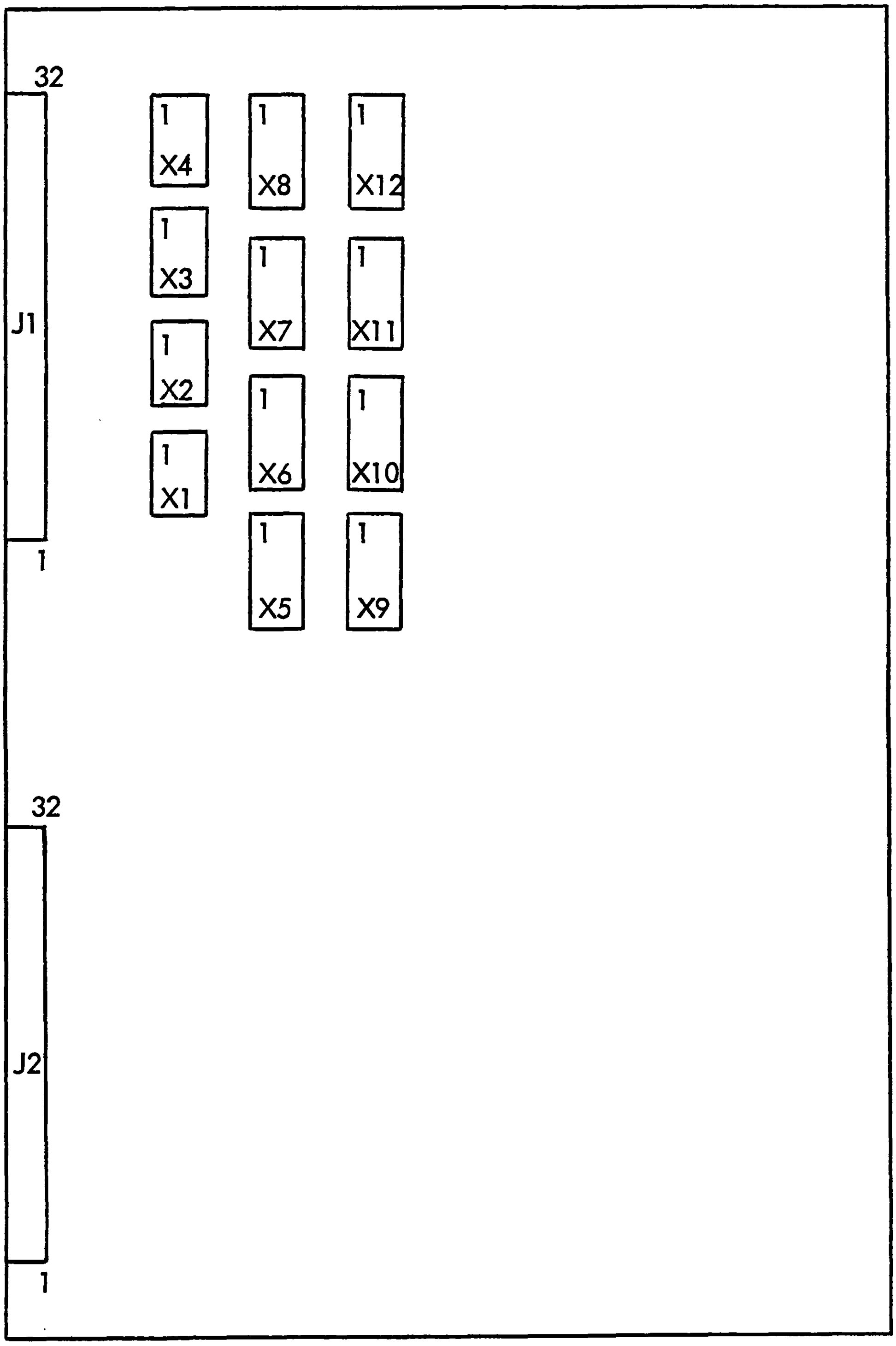


Fig. 12.2. Layout of 4K Memory Board.

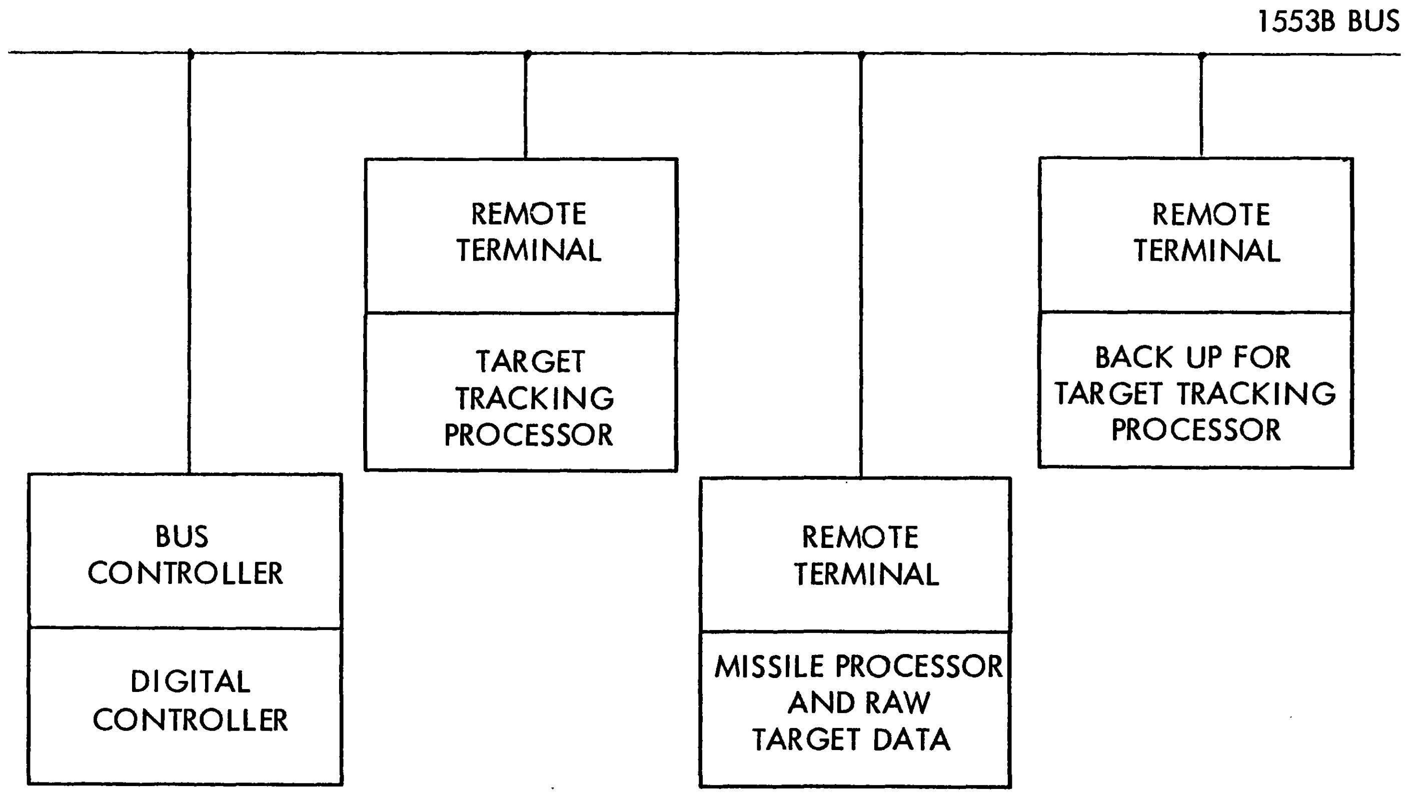

1553B BUS
BUS CONTROLLER
DIGITAL CONTROLLER
REMOTE TERMINAL
TARGET TRACKING PROCESSOR
REMOTE TERMINAL
MISSILE PROCESSOR AND RAW TARGET DATA
REMOTE TERMINAL
BACK UP FOR TARGET TRACKING PROCESSOR

Fig. 12.3. System Configuration for Remote Terminal Failure.

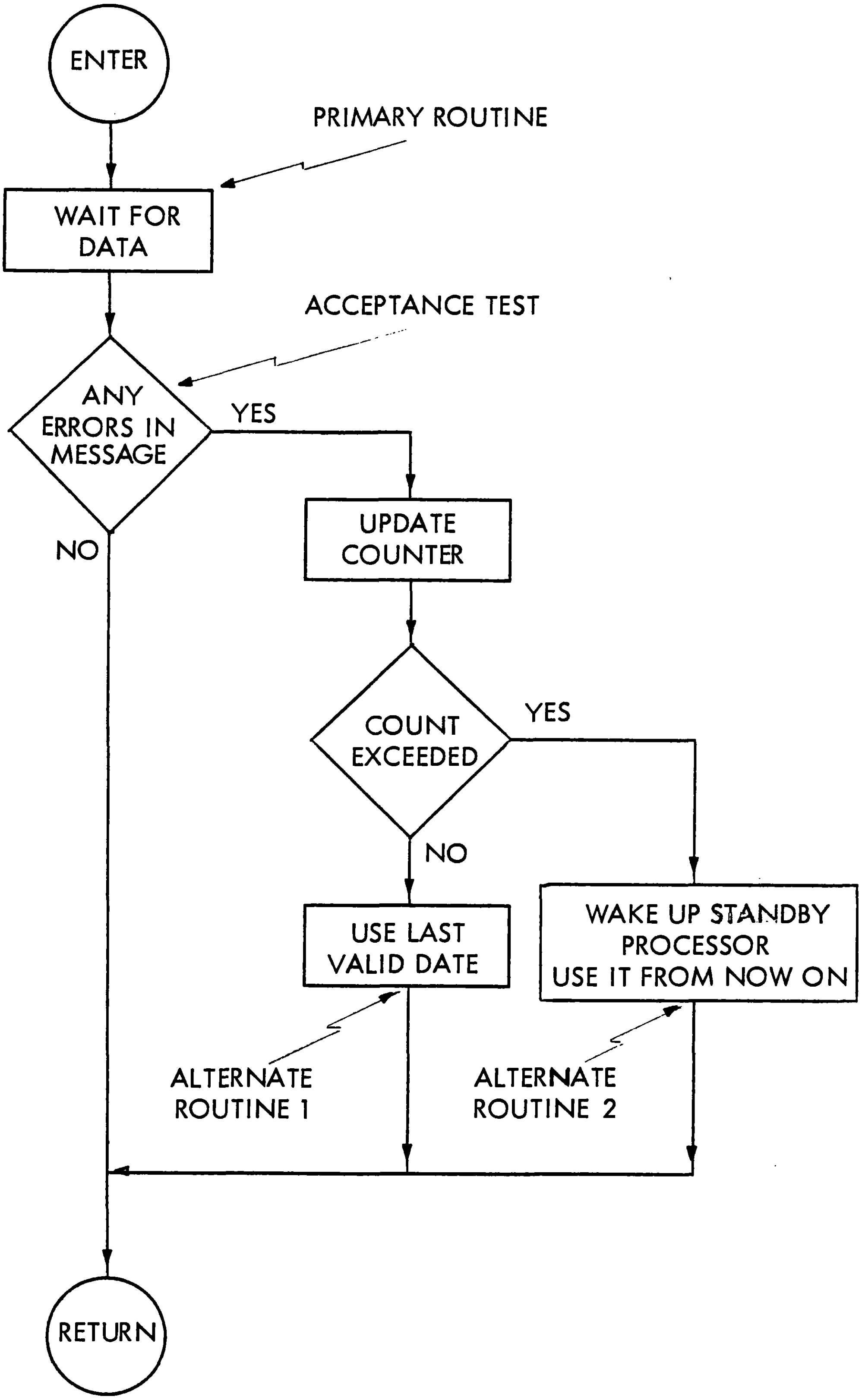


Fig. 12.4. Schematic of Task Swapping.

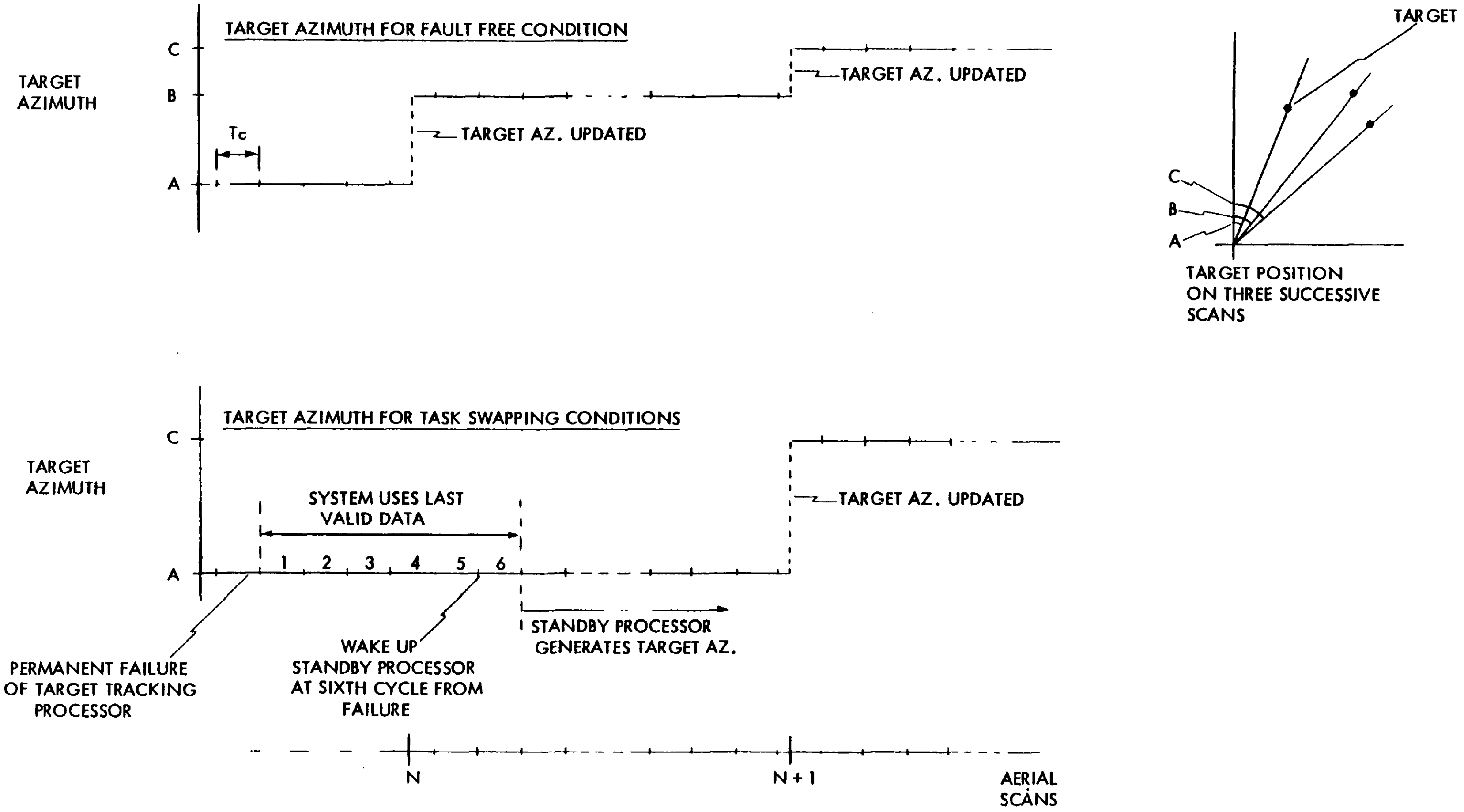

TARGET AZIMUTH FOR FAULT FREE CONDITION
TARGET AZIMUTH
C
B
A
Tc
TARGET AZ. UPDATED
TARGET AZ. UPDATED
TARGET
C
B
A
TARGET POSITION ON THREE SUCCESSIVE SCANS
TARGET AZIMUTH FOR TASK SWAPPING CONDITIONS
TARGET AZIMUTH
C
A
SYSTEM USES LAST VALID DATA
1
2
3
4
5
6
TARGET AZ. UPDATED
STANDBY PROCESSOR GENERATES TARGET AZ.
PERMANENT FAILURE OF TARGET TRACKING PROCESSOR
WAKE UP STANDBY PROCESSOR AT SIXTH CYCLE FROM FAILURE
N
N + 1
AERIAL SCÁNS

Fig. 12.5. Failure of Target Tracking Processor.

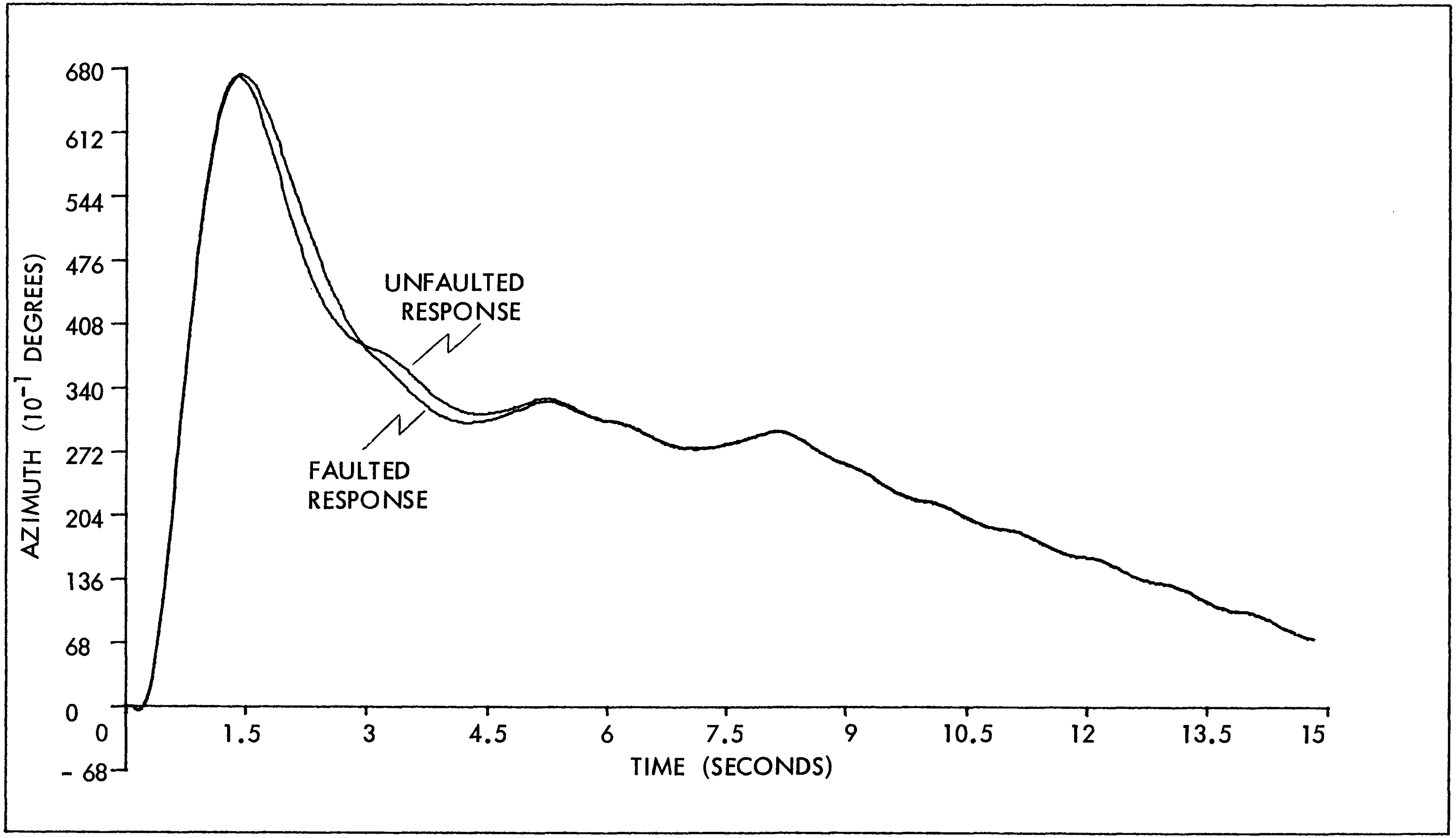


Fig. 12.6. Remote Terminal Failure in Gathering Phase.

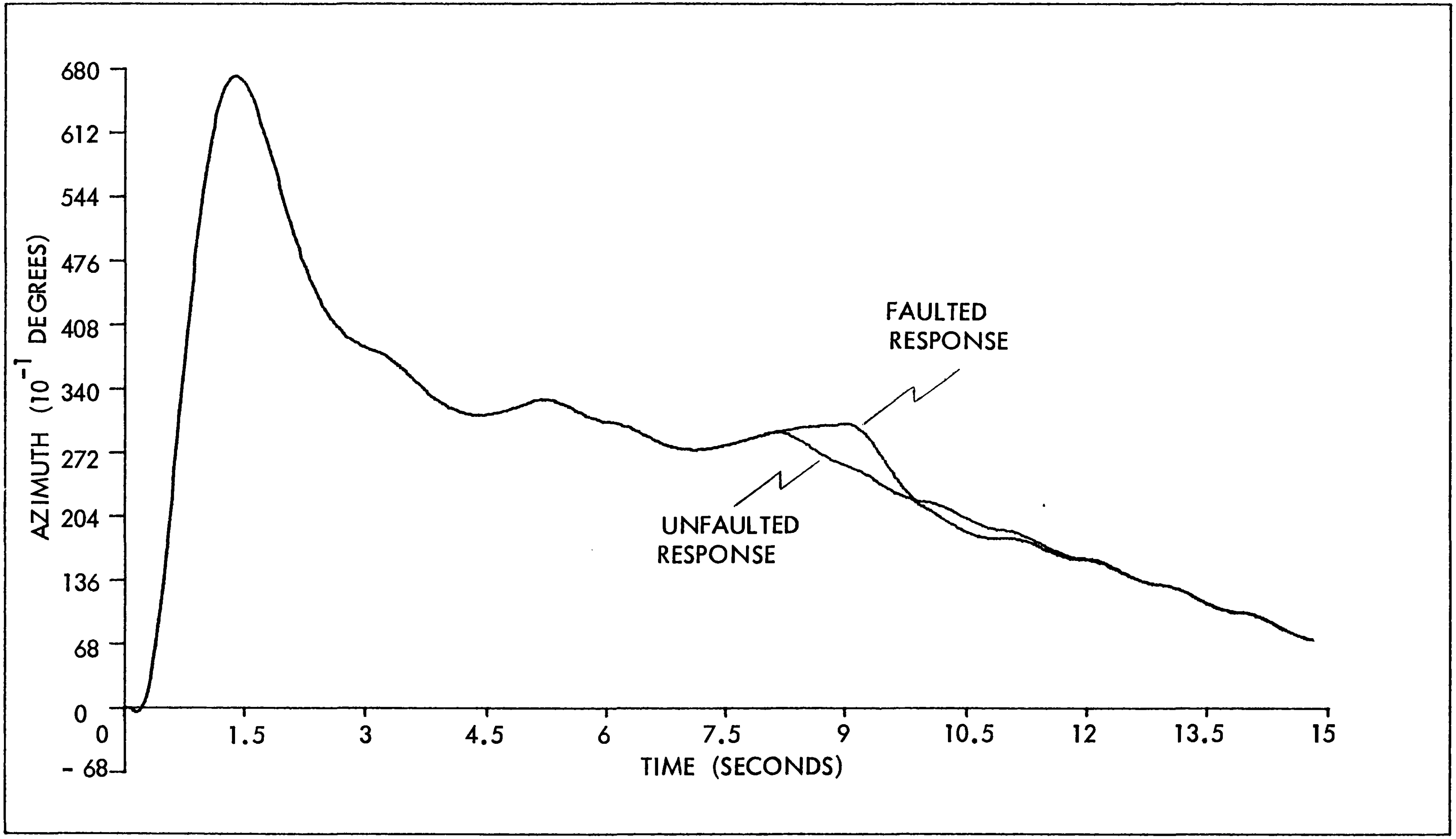


Fig.12.7. Remote Terminal Failure in Terminal Phase.

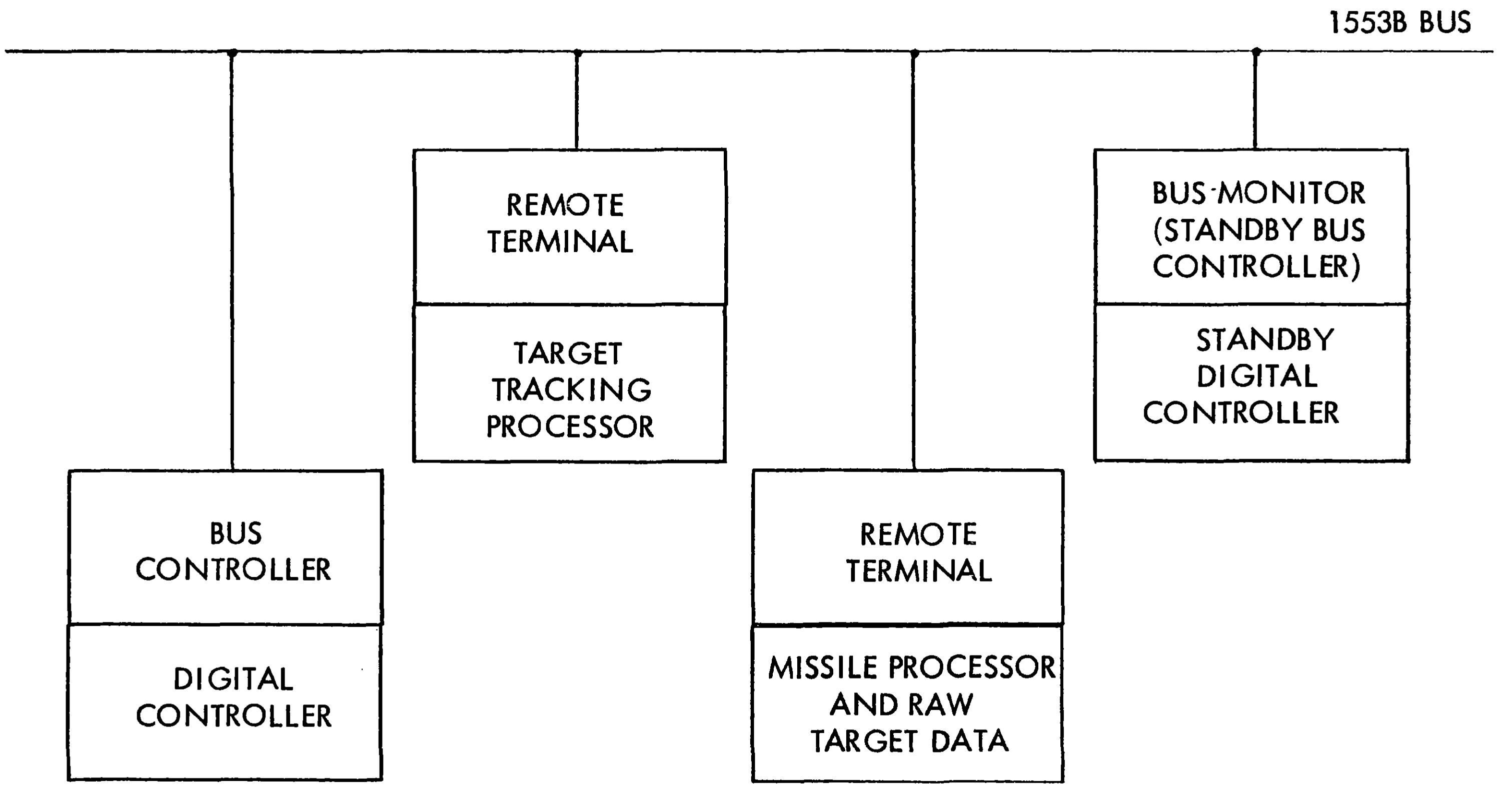
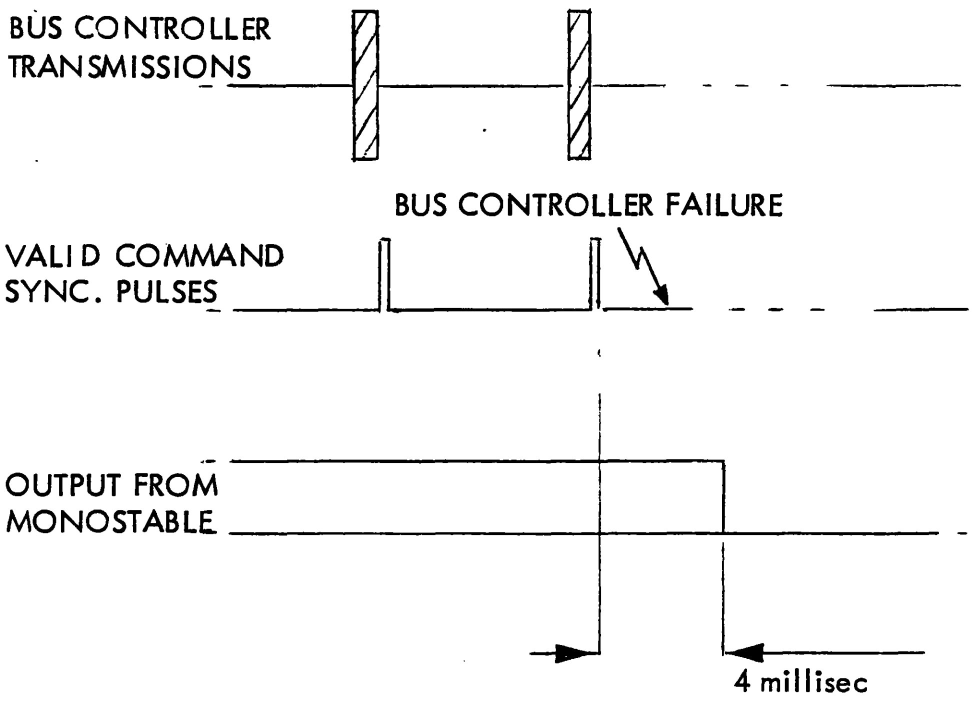

1553B BUS
BUS CONTROLLER
DIGITAL CONTROLLER
REMOTE TERMINAL
TARGET TRACKING PROCESSOR
REMOTE TERMINAL
MISSILE PROCESSOR AND RAW TARGET DATA
BUS-MONITOR (STANDBY BUS CONTROLLER)
STANDBY DIGITAL CONTROLLER

Fig. 12.8. System Configuration for Bus Controller Failure.



NOTE: TIME PERIODS NOT TO SCALE

Fig.12.9. Bus Inactivity Detection.

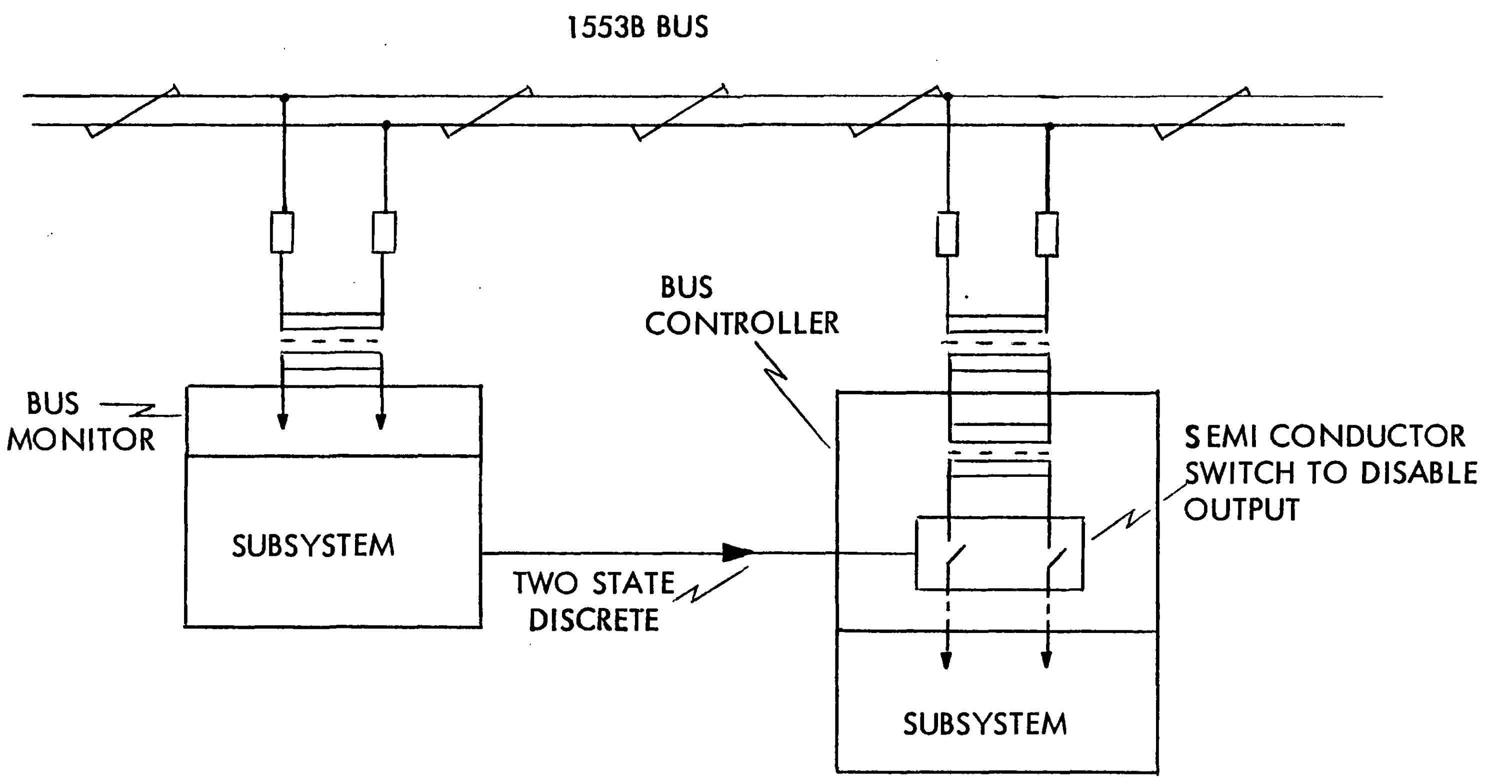


Fig. 12.10. Use of Discrete to Disable Failed Bus Controller.

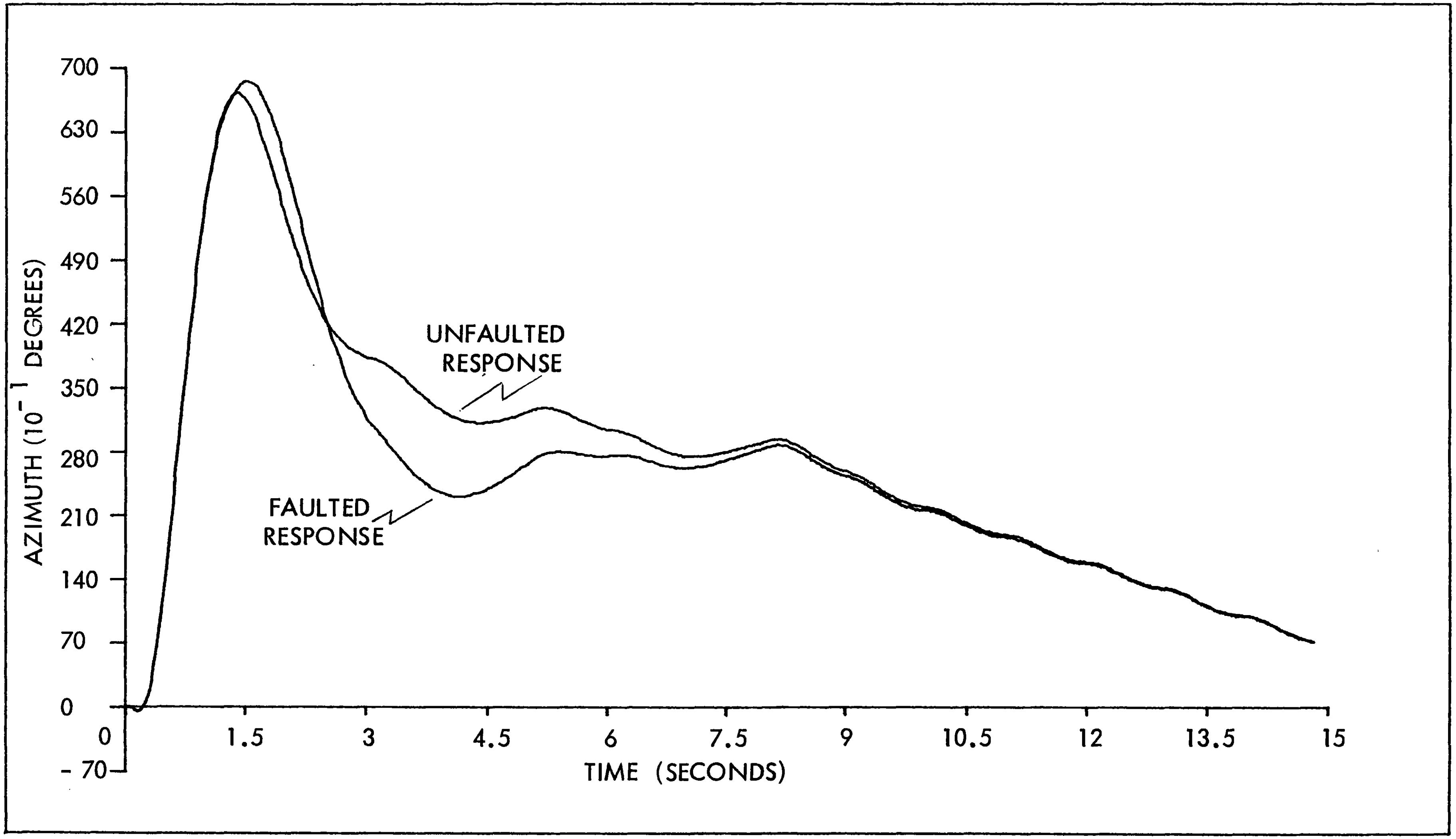


Fig.12.11. Bus Controller Failure in Gathering Phase (1 second).

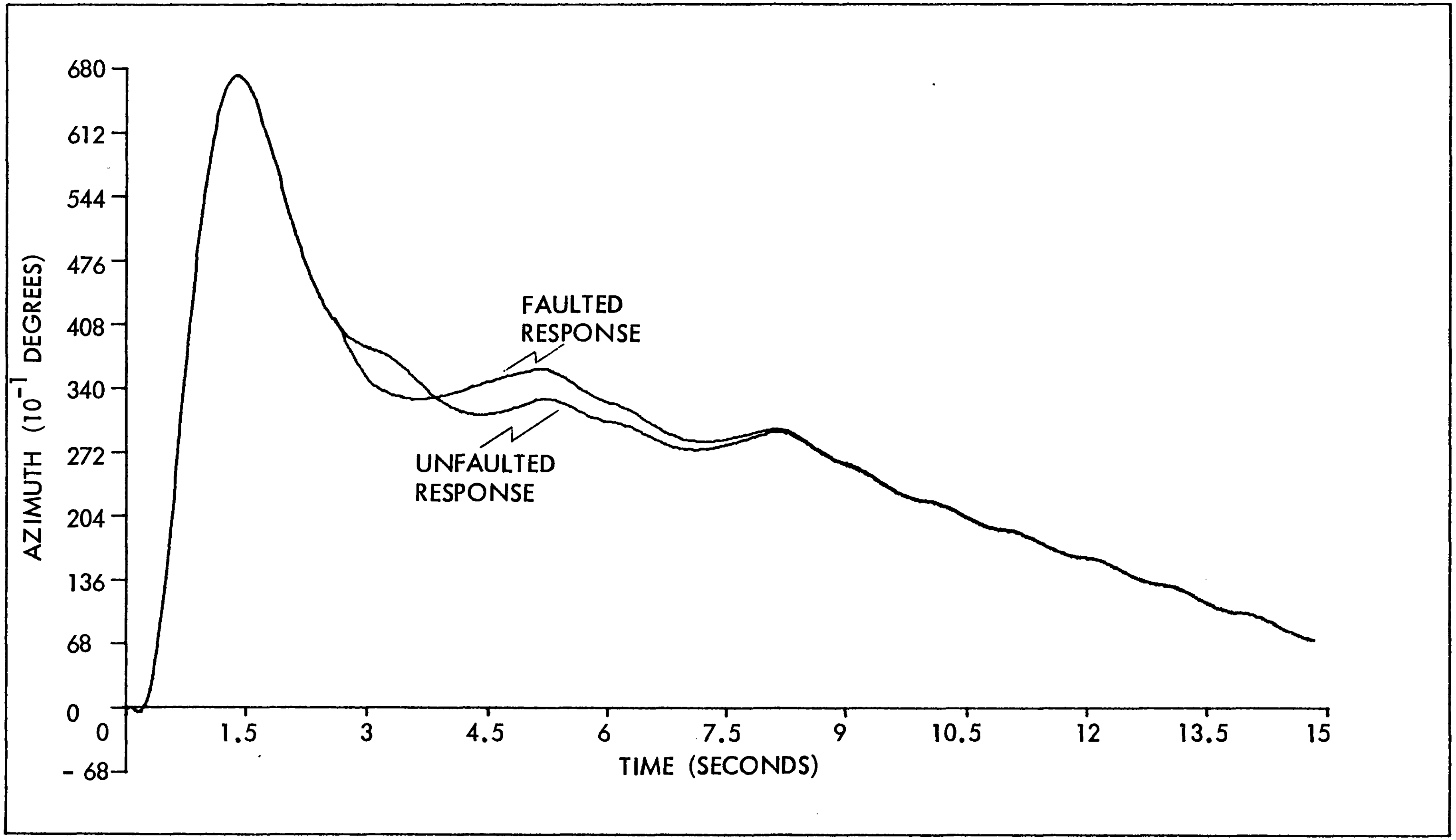


Fig. 12.12. Bus Controller Failure in Gathering Phase (2 seconds).

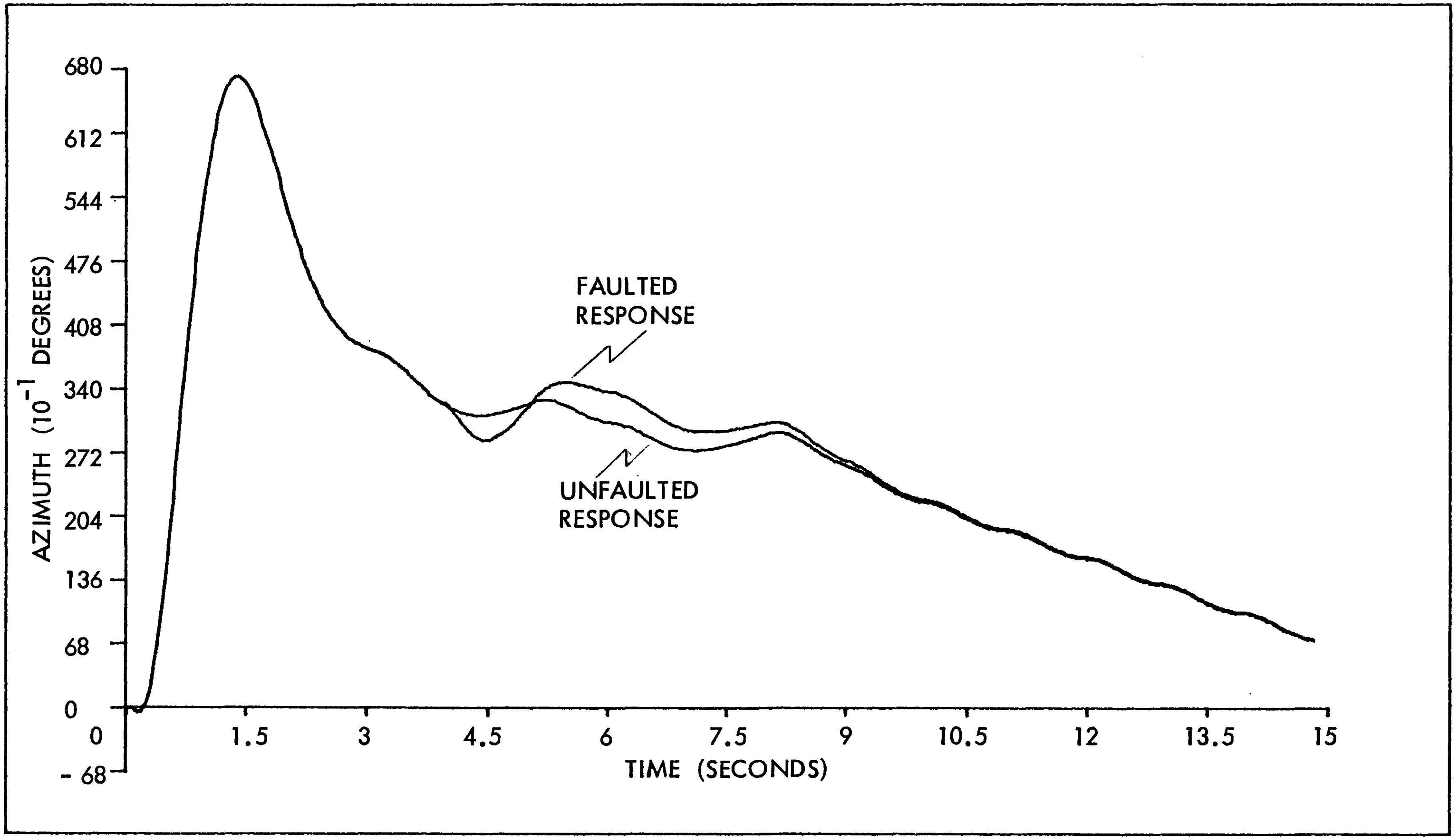


Fig. 12.13. Bus Controller Failure in Gathering Phase (4 seconds).

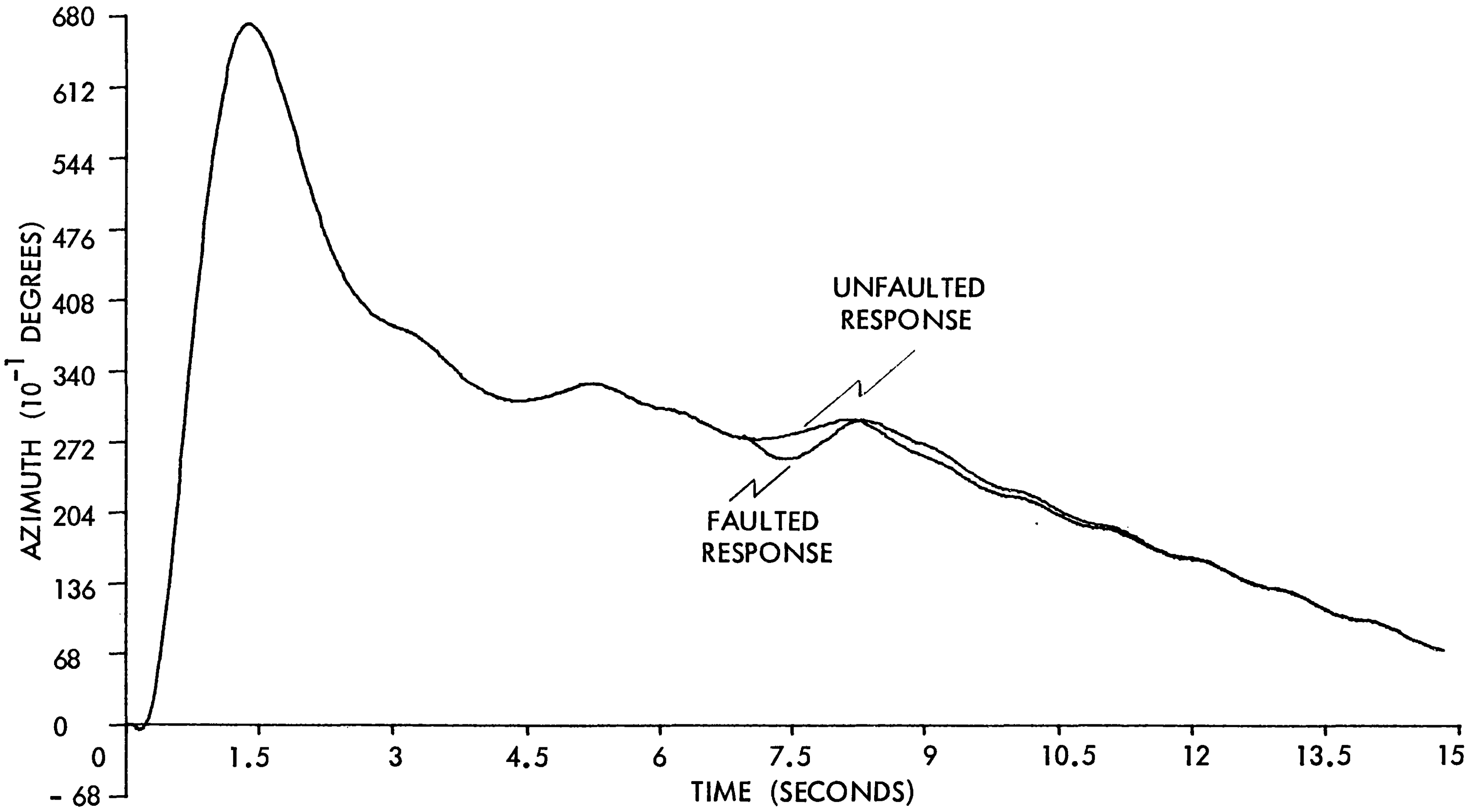


Fig.12.14. Bus Controller Failure in Terminal Phase (7 seconds).

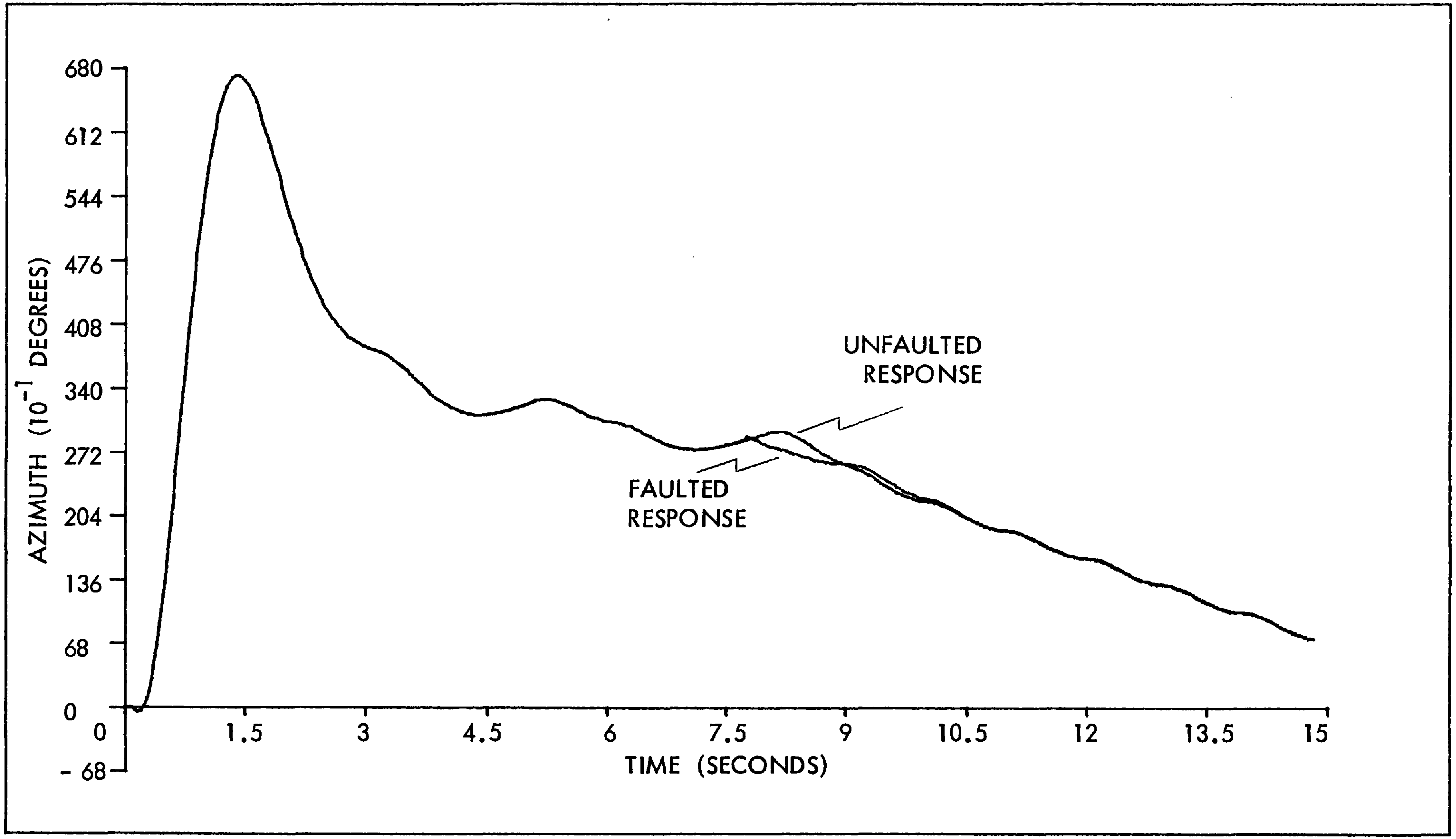


Fig.12.15. Bus Controller Failure in Terminal Phase (8 seconds).

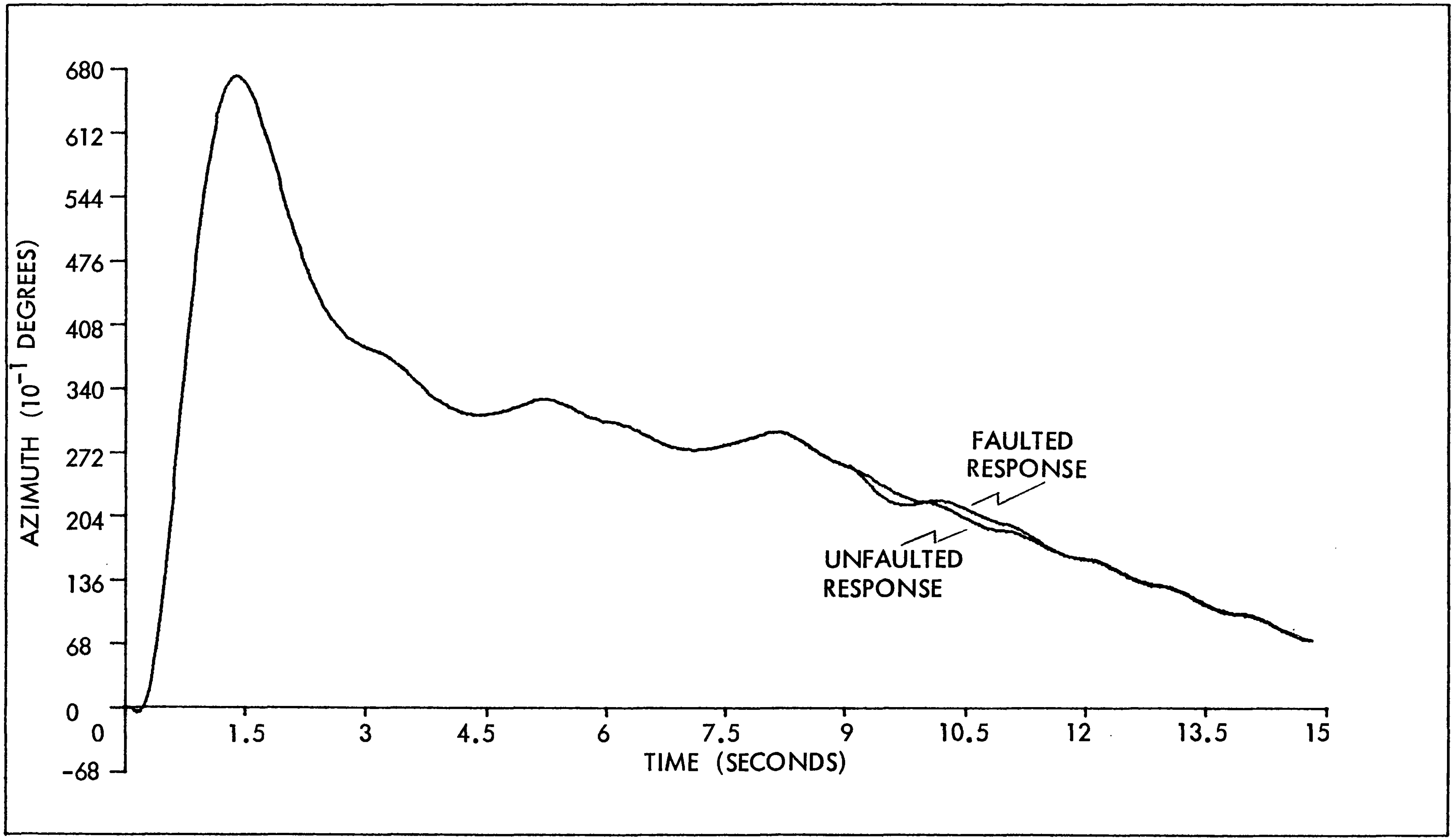


Fig. 12.16. Bus Controller Failure in Terminal Phase (9 seconds).

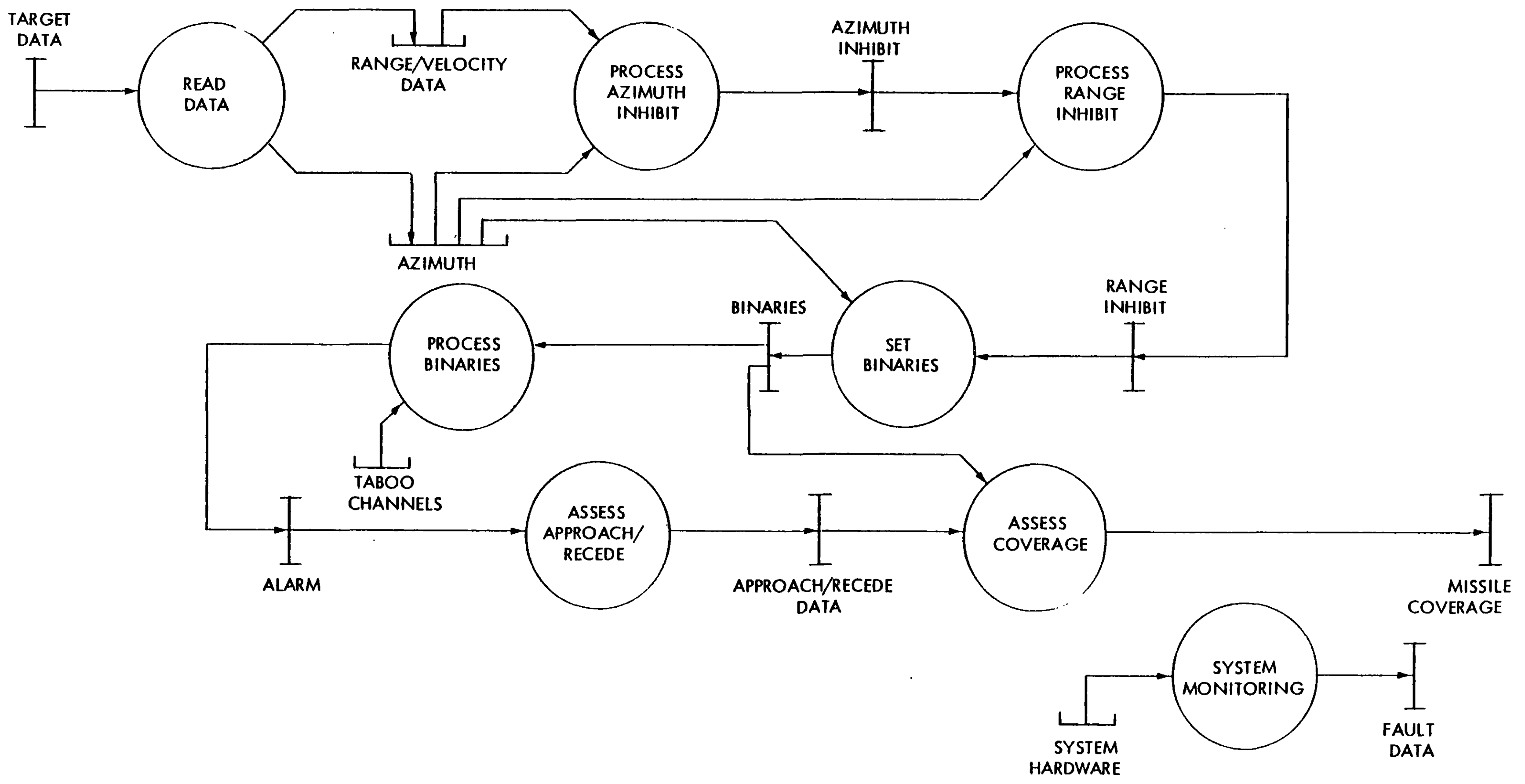


Fig. 14.1. MASCOT Diagram for Target Tracking Process.

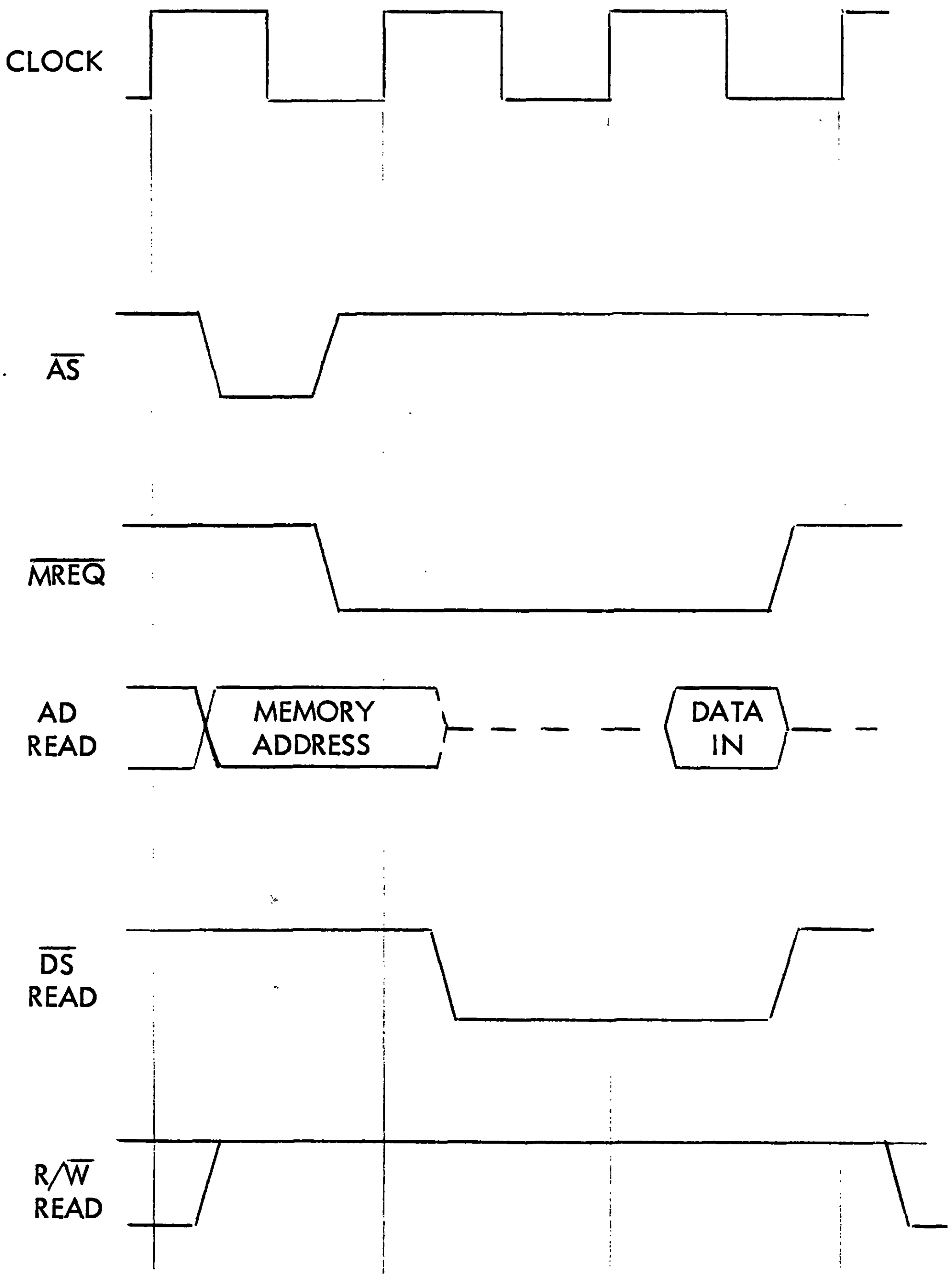


Fig.B.1. Z8000 Memory Read Cycle.

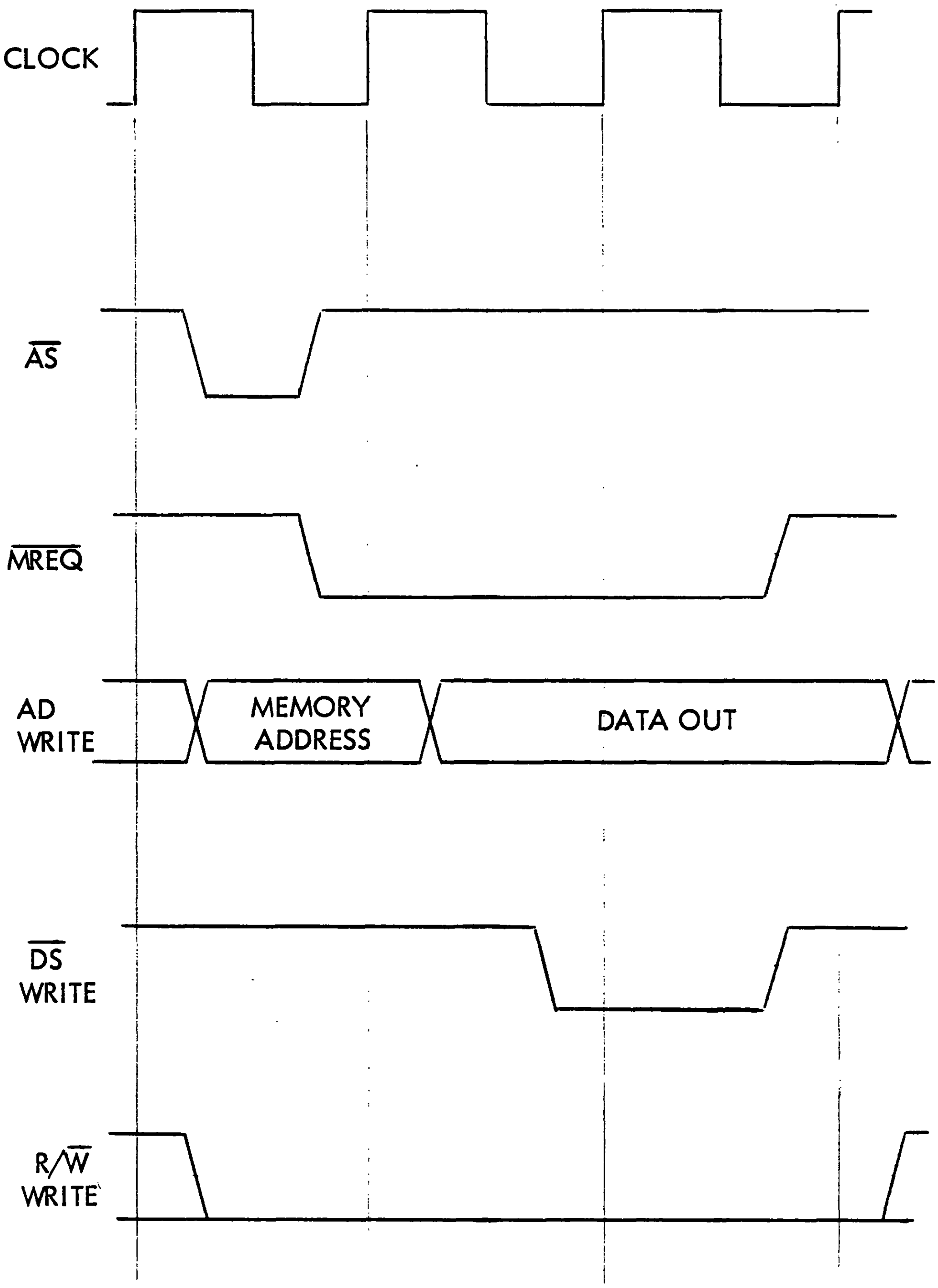


Fig.B.2. Z8000 Memory Write Cycle.

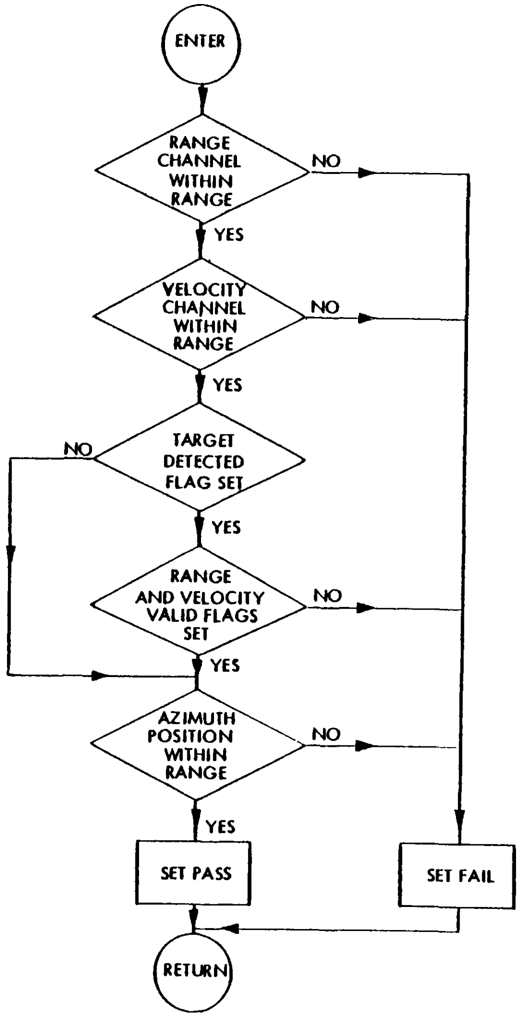

ENTER
RANGE CHANNEL WITHIN RANGE
NO
YES
VELOCITY CHANNEL WITHIN RANGE
NO
YES
TARGET DETECTED FLAG SET
NO
YES
RANGE AND VELOCITY VALID FLAGS SET
NO
YES
AZIMUTH POSITION WITHIN RANGE
NO
YES
SET PASS
SET FAIL
RETURN

Fig. E.1. Acceptance Test for Read Routine.

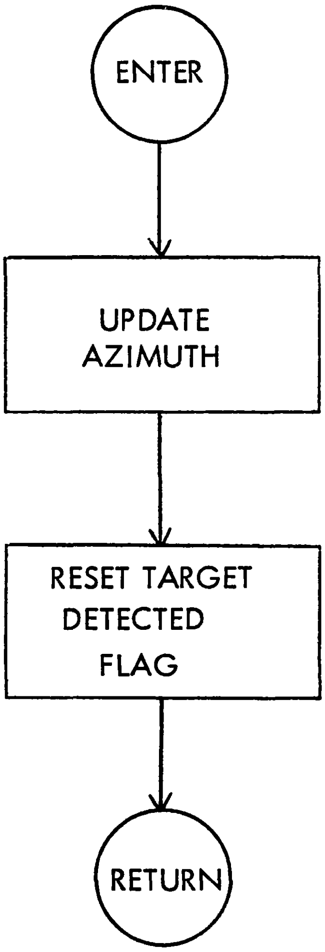


Fig. E.2. Read Alternate Routine.

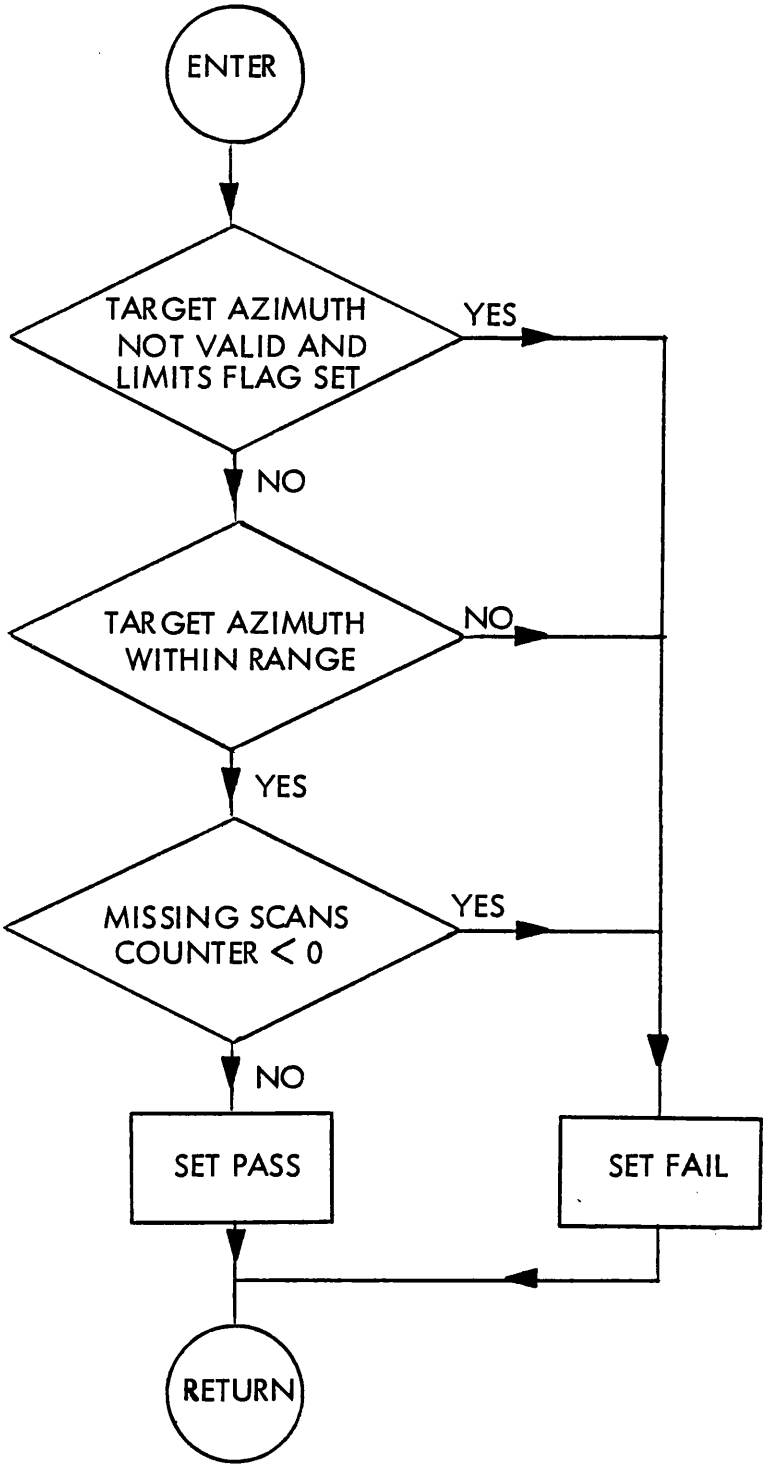

ENTER
TARGET AZIMUTH NOT VALID AND LIMITS FLAG SET
YES
NO
TARGET AZIMUTH WITHIN RANGE
NO
YES
MISSING SCANS COUNTER < 0
YES
NO
SET PASS
SET FAIL
RETURN

Fig. E. 3. Acceptance Test for Azimuth Inhibit.

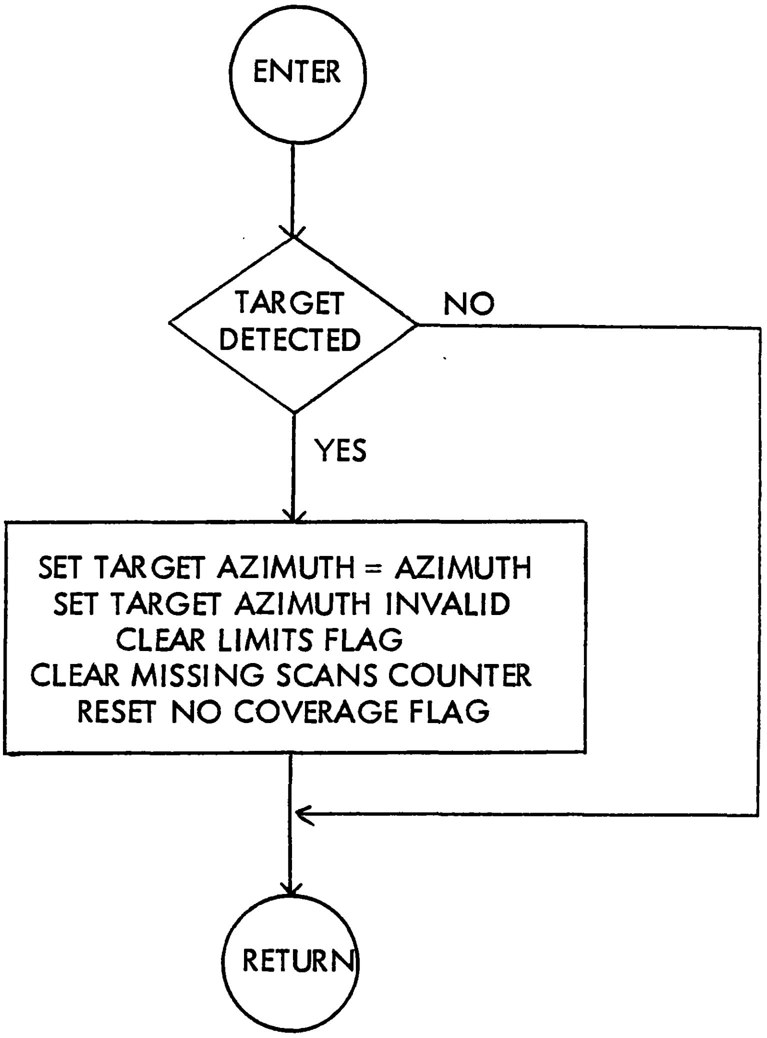

ENTER
TARGET DETECTED
NO
YES
SET TARGET AZIMUTH = AZIMUTH
SET TARGET AZIMUTH INVALID
CLEAR LIMITS FLAG
CLEAR MISSING SCANS COUNTER
RESET NO COVERAGE FLAG
RETURN

Fig. E. 4. Azimuth Inhibit Alternate Routine.

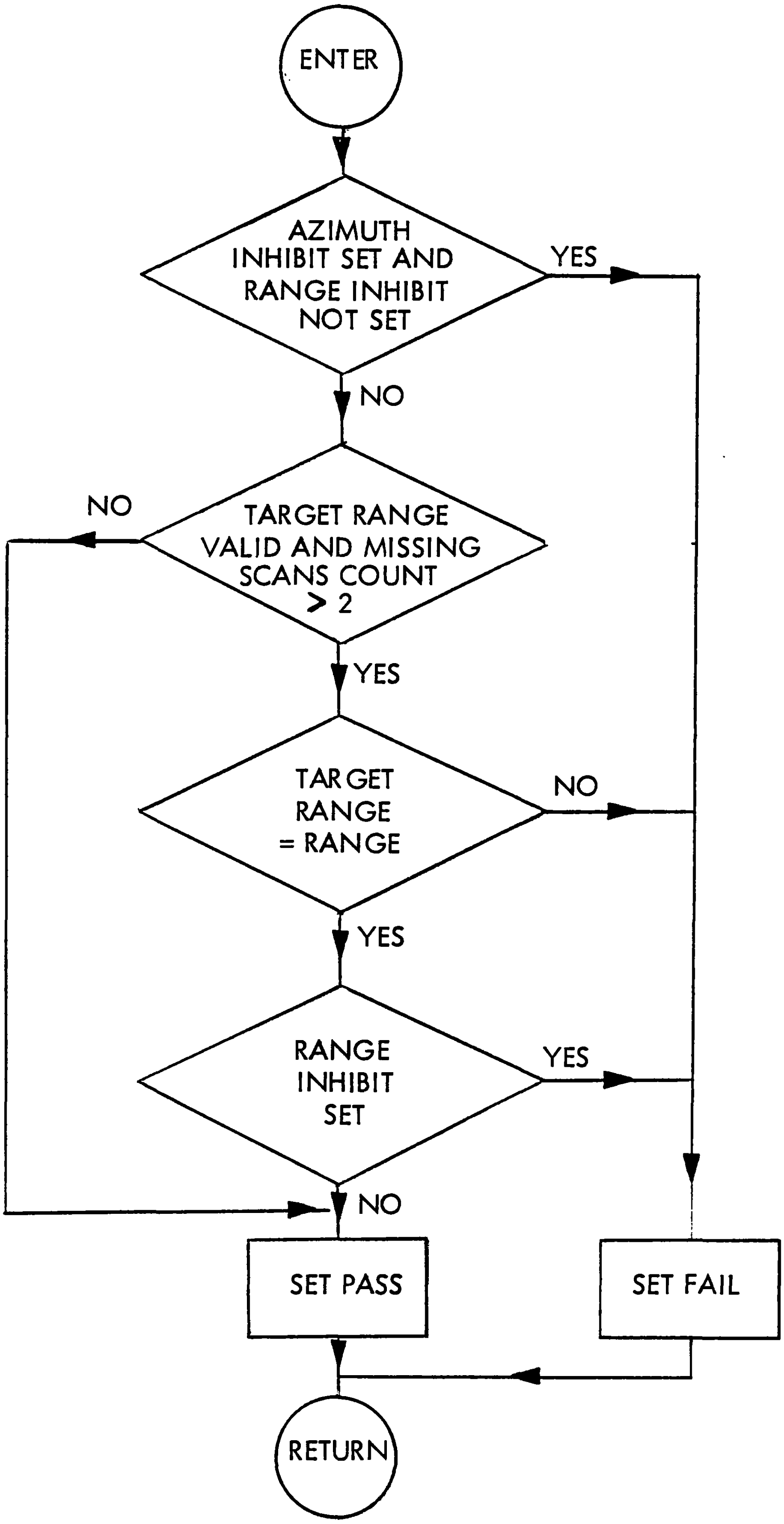

ENTER
AZIMUTH INHIBIT SET AND RANGE INHIBIT NOT SET
YES
NO
TARGET RANGE VALID AND MISSING SCANS COUNT > 2
NO
YES
TARGET RANGE = RANGE
NO
YES
RANGE INHIBIT SET
YES
NO
SET PASS
SET FAIL
RETURN

Fig. E.5. Acceptance Test for Range Inhibit.

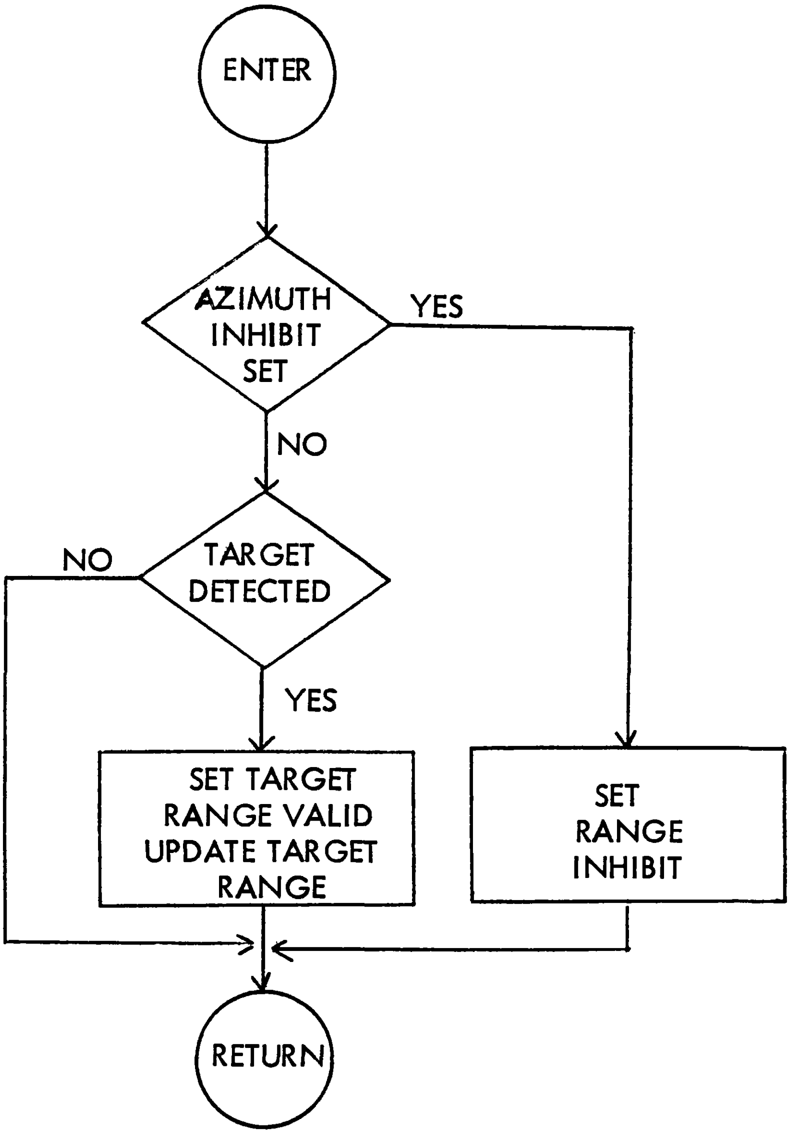


Fig. E. 6. Range Inhibit Alternate Routine.

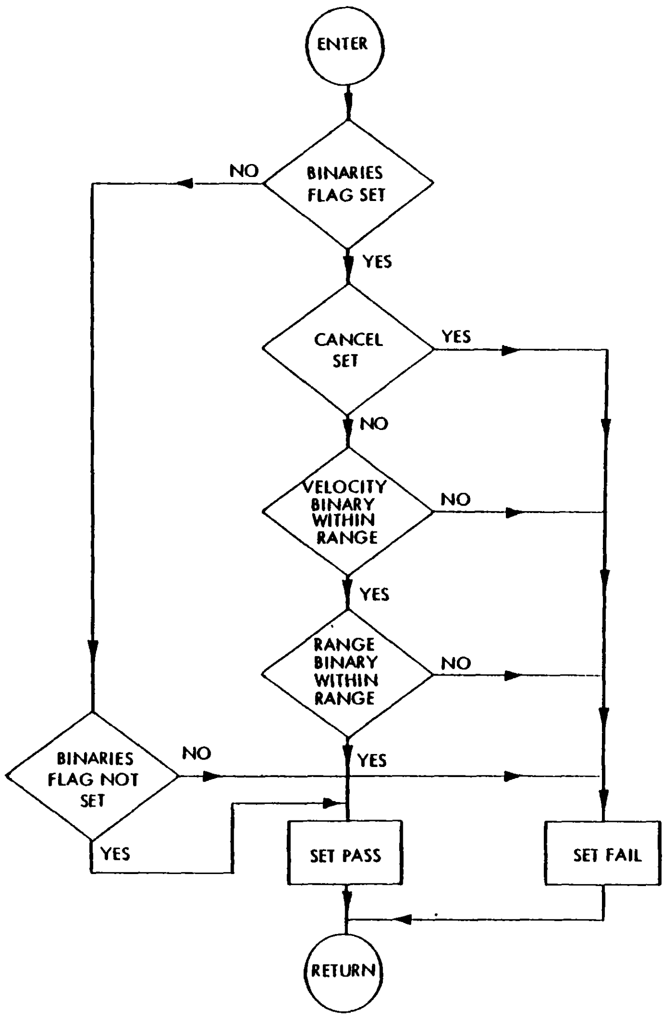


Fig. E.7. Acceptance Test for Set Binaries.

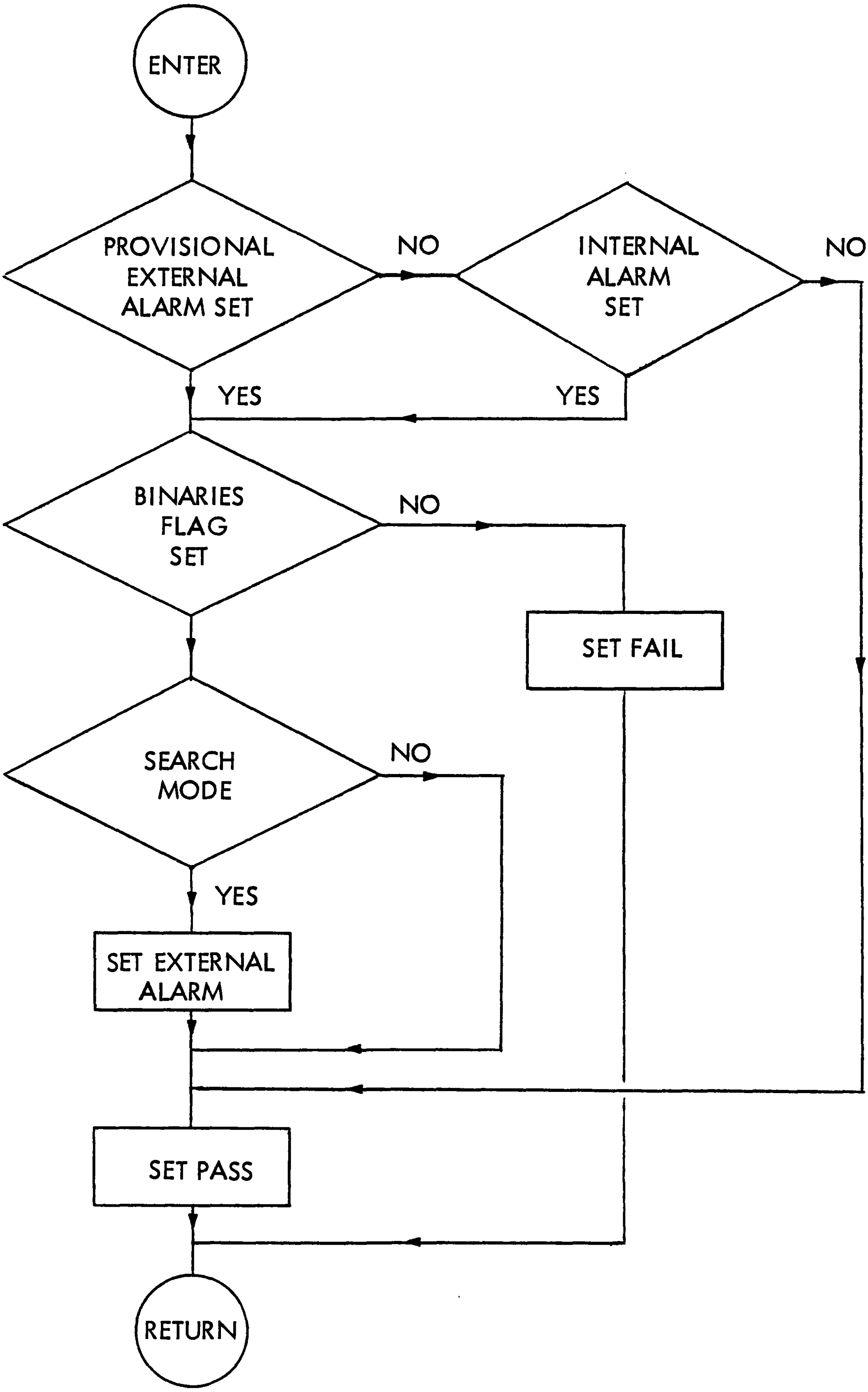


Fig. E. 8.   Acceptance Test for Process Binaries.

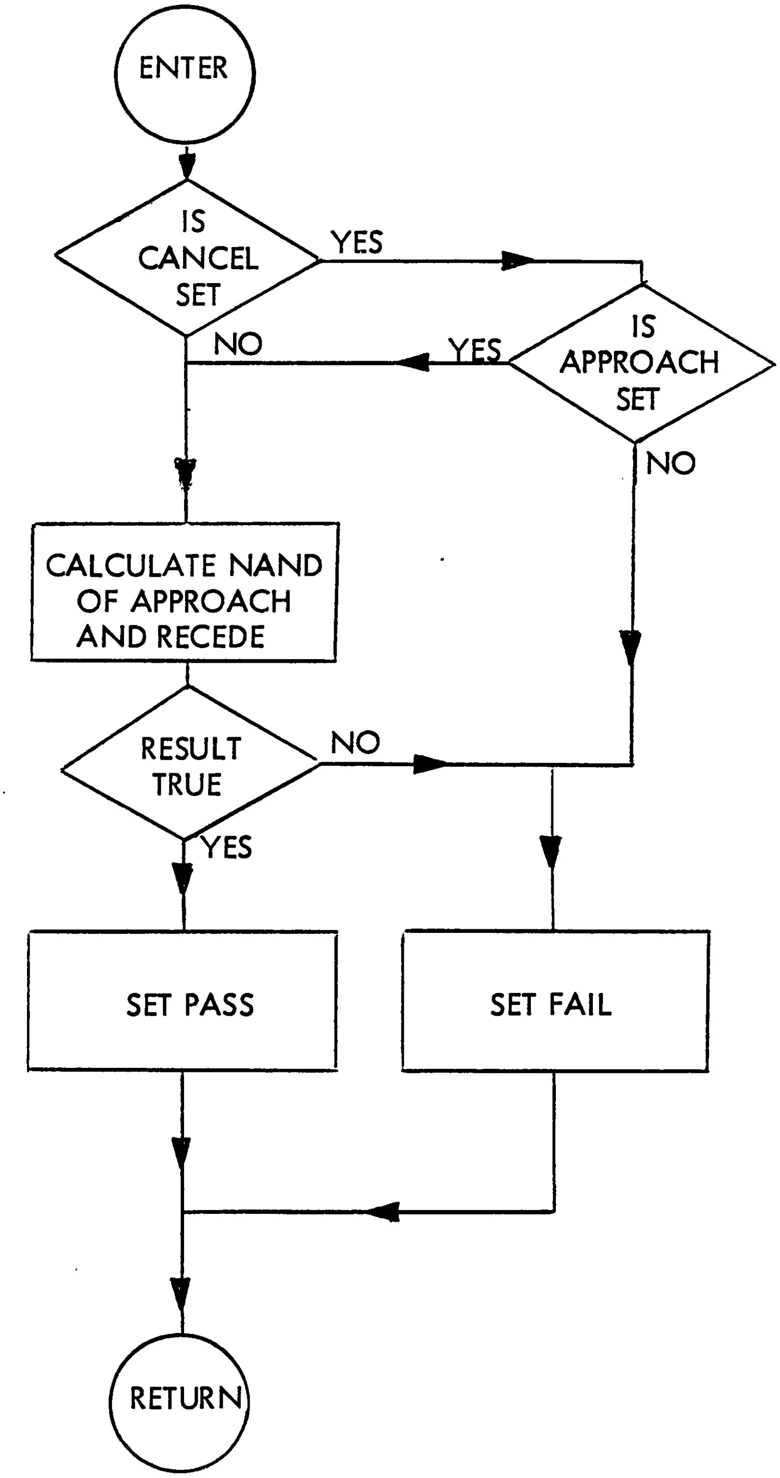


Fig. E.9. Acceptance Test for Approach/Recede Assessment.

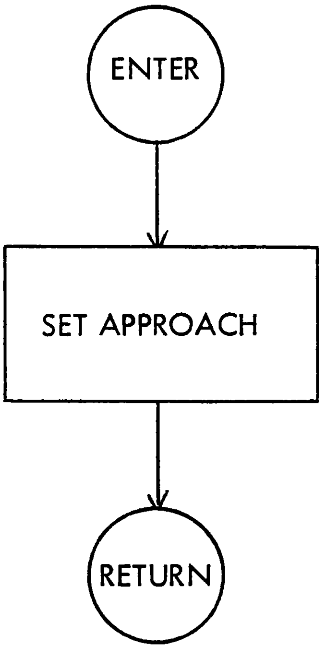


Fig. E.10. Approach/Recede Assessment Alternate Routine.

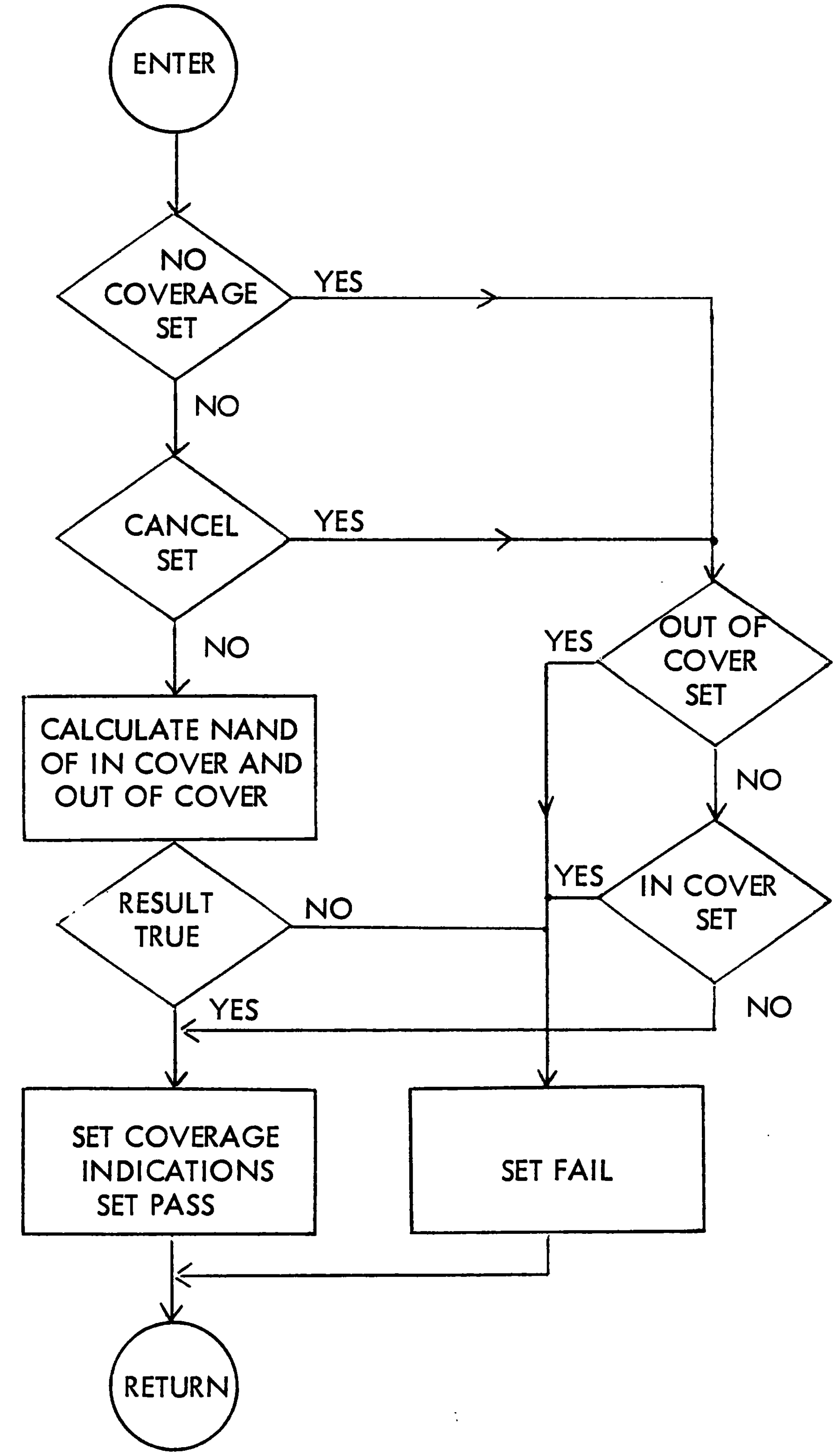


Fig. E.11. Acceptance Test for Coverage Assessment.

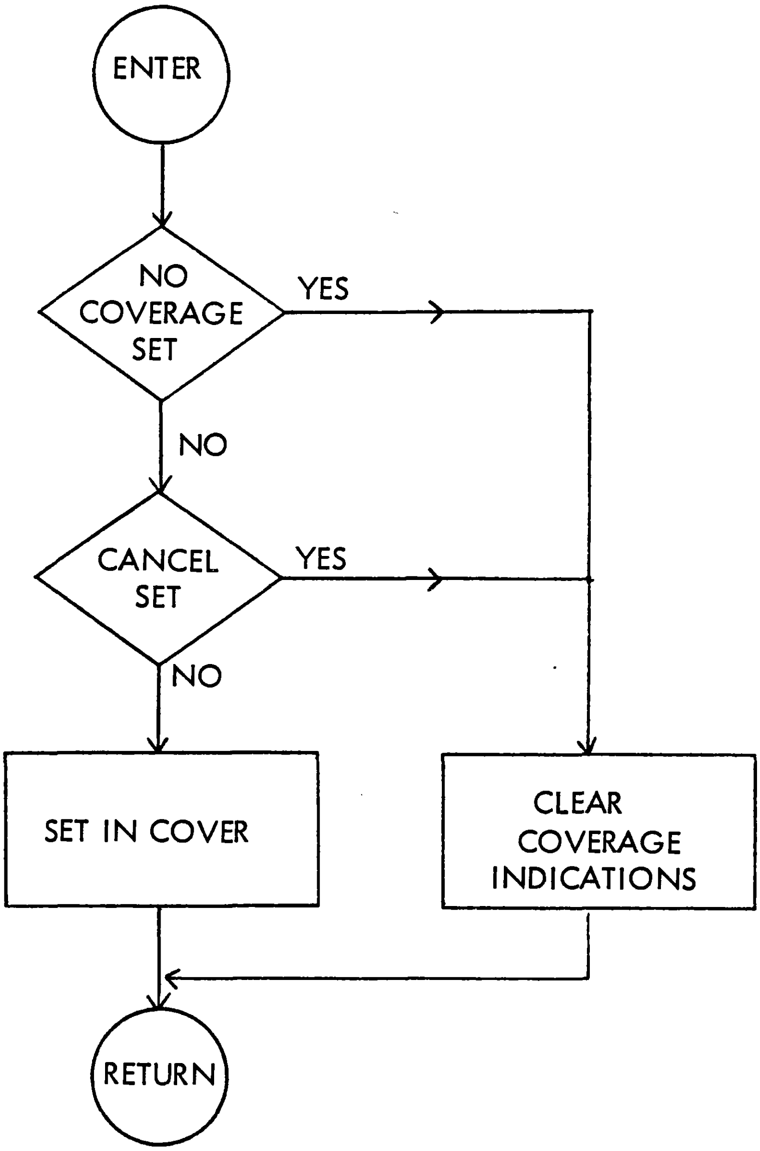


Fig. E. 12. Coverage Assessment Alternate Routine.

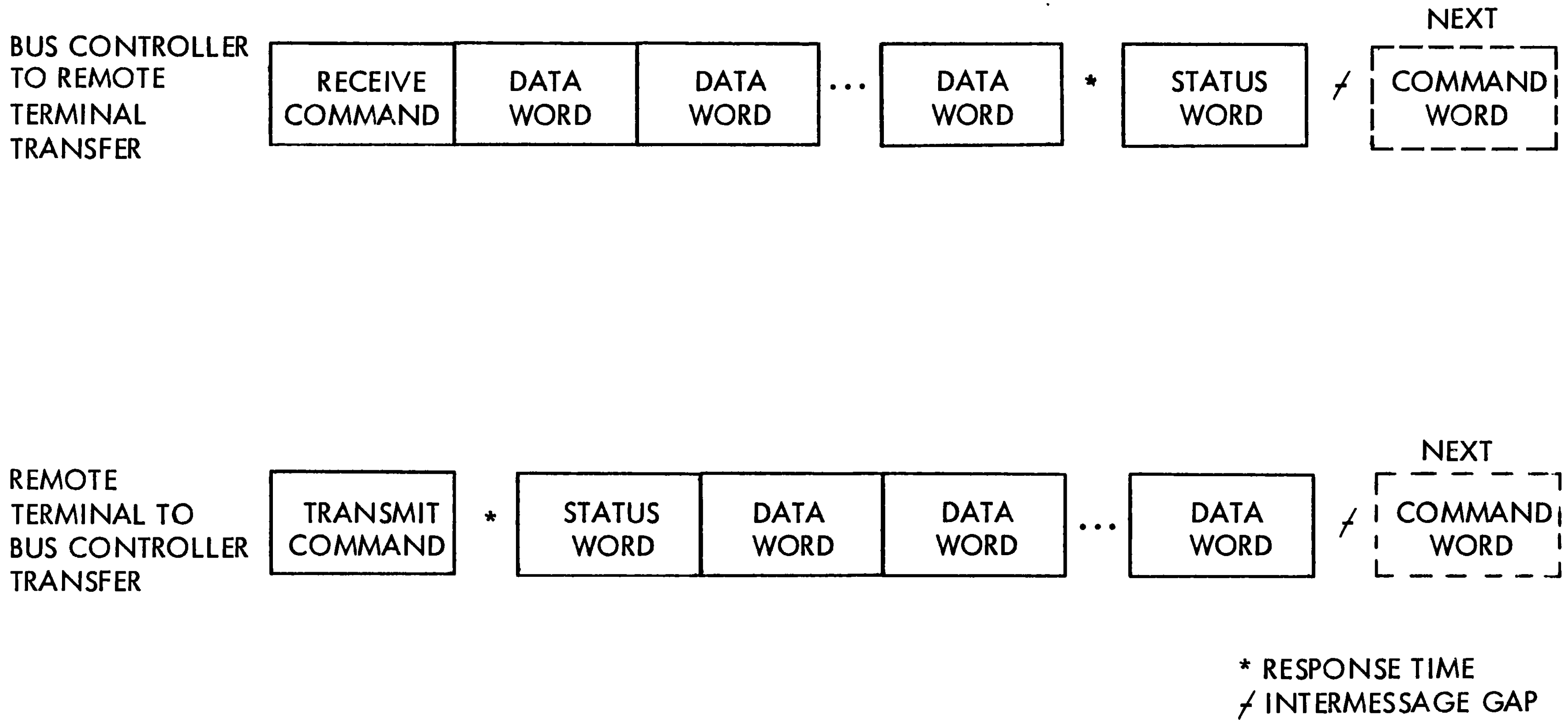
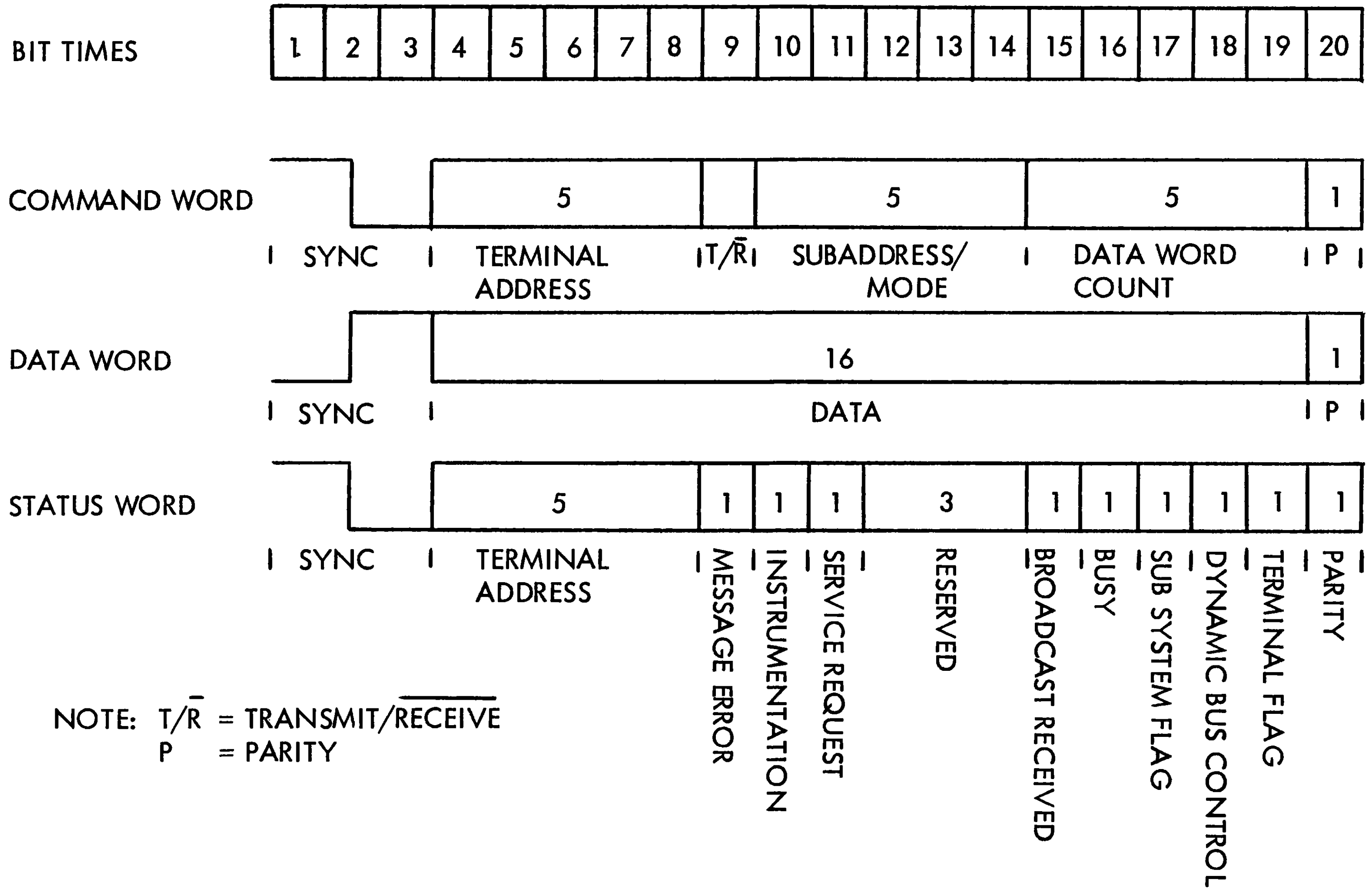


Fig.F.1. 1553B Message Formats.



NOTE: T/$\bar{R}$ = TRANSMIT/$\overline{\text{RECEIVE}}$
P = PARITY

Fig.F.2. 1553B Word Formats.

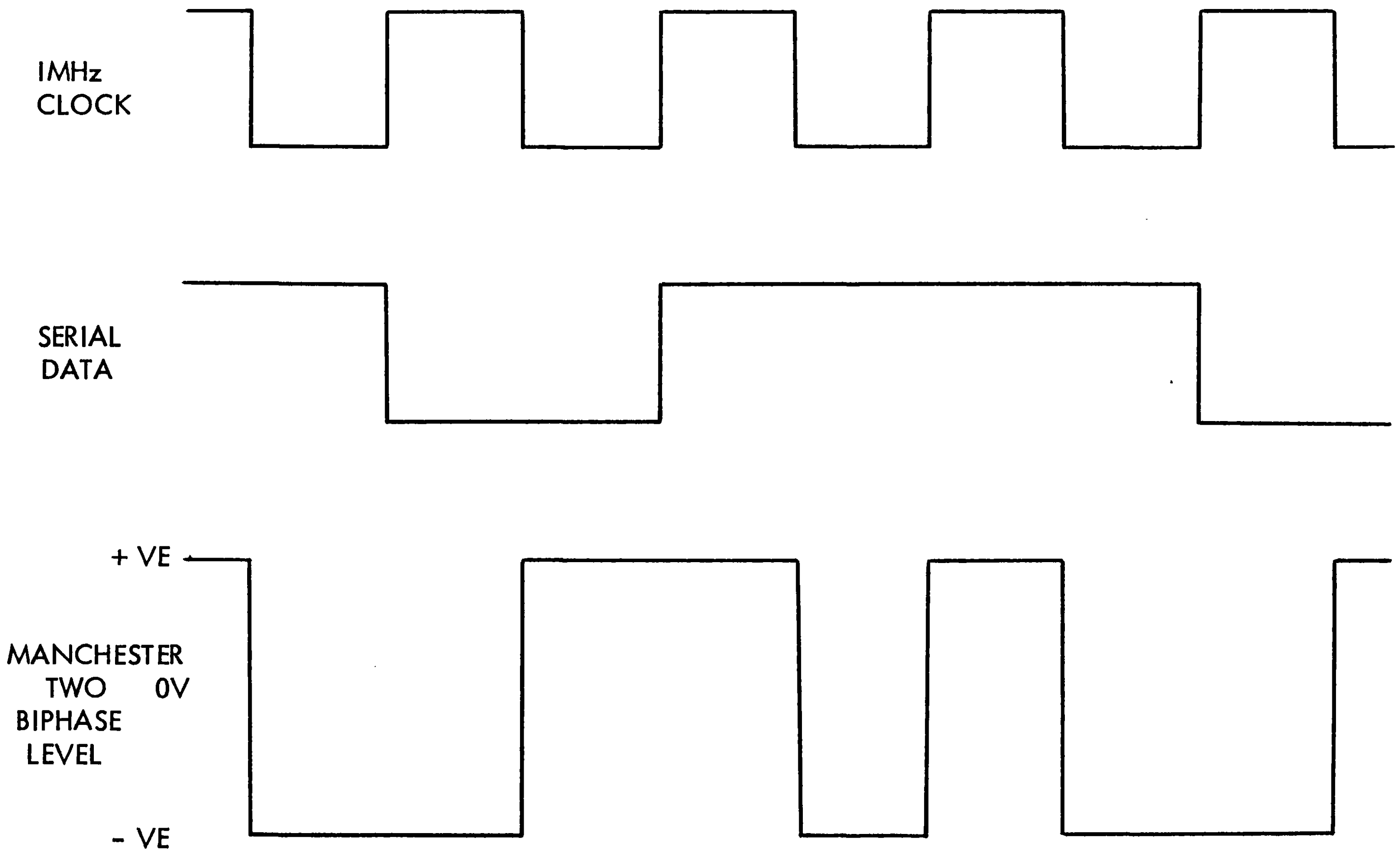


Fig.F.3. Data Encoding.

Range Gate No.	Velocity Gate No.	Approach	Recede
1	1	180	0
1	2	160	0
1	3	50	0
1	4	20	150
2	1	105	0
2	2	90	0
2	3	60	0
2	4	20	50
3	1	50	0
3	2	52	0
3	3	40	0
3	4	25	0
4	1	0	0
4	2	34	0
4	3	30	0
4	4	24	0
5	1	0	0
5	2	12	0
5	3	20	0
5	4	15	0
6	1	0	0
6	2	0	0
6	3	13	0
6	4	13	0

Table 4.1. Angular Rate Information.

Integrated Circuits

X1	74LS04	X2	74LS20	X3	74LS32
X4	74LS32	X5	74LS74		
X9 – X43	MM2102AN				

Table 6.1. Parts List of Error Correcting Memory Board 1.

Integrated Circuits

X1	74LS86	X2	74LS86	X3	74LS86
X4	74LS86	X5	74LS157	X6	74LS157
X7	74LS157	X8	74LS157	X9	74LS280
X10	74LS280	X11	74LS280	X12	74LS86
X13	74LS154	X14	74LS154	X15	74LS240
X16	74LS240	X17	74LS280	X18	74LS04
X19	74LS126	X20	74LS126	X21	74LS280
X22	74LS244	X23 — X32	MM2102AN		
X33	74LS00				

Table 6.2. Parts List of Error Correcting Memory Board 2.

Integrated Circuits

X1	74276	X2	74LS32	X3	74LS04
X4	74LS32	X5	74LS08	X6	74LS125
X7	74LS27	X8	74LS20	X9	74LS138
X10	74LS00	X11	74LS74	X12	74LS00
X13	74LS93	X14	74LS00	X15	74LS74
X16	4020B	X17	74LS74	X18	74LS08
X19	74LS32	X20	74LS374	X21	74LS151
X22	74LS138				

Resistors (± 5%)

R1 — R12 1K

Table 6.3. Parts List of Input/Output Board.

Integrated Circuits

X1	74LS244	X2	74LS244	X3	74LS244
X4	74LS244	X5	74LS193	X6	74LS02
X7	74LS374	X8	74LS374	X9	74LS195

Resistor (± 5%)

R1	1K

Table 6.4. Parts List of Buffer Card.

Integrated Circuits

X1	74LS00	X2	74LS00	X3	74LS04
X4	74LS125	X5	74LS10	X6	74LS00
X7	74LS00	X8	74LS00	X9	74LS04
X10	74LS125	X11	74LS32	X12	74LS10
X14	74LS123				

Resistors (± 5%)

R1	5.1K	R2	50K POT	R3	5.1K
R4	50K POT	R5	1K		

Capacitors (± 20%)

C1	22pF	C2	22pF

Note: R2 and R4 mounted on front panel of expansion box.

Table 6.5. Parts List of Fault Injection Logic.

Connections of P1, P2 and P3 on CPU Card determine baud rate as follows:

9600 baud	-	all open
2400 baud	-	connect P1 to P3
300 baud	-	connect P2 to P3
100 baud	-	connect P1 to P2 to P3

Table 10.1. Baud Rate Selection.

Integrated Circuits

X1	74LS32	X2	74LS10	X3	74LS244
X4	74LS44	X5	74LS245	X6	74LS245
X7	74LS374	X8	74LS374	X9	74LS74
X10	74LS139	X11	74LS138	X12	74LS27
X13	74LS32	X14	MM2114	X15	MM2114
X16	MM2114	X17	MM2114	X18	74LS244
X19	74LS164	X20	74LS02	X21	25LS2521
X22	74LS32	X23	MM2114	X24	MM2114
X25	MM2114	X26	MM2114	X27	Am9551
X28	74LS30	X29	25LS2521	X30	74LS04
X31	74LS00	X32	AmZ8002	X33	Not Used
X34	74LS138	X35	74LS138	X36	74LS74
X37	74LS08	X38	74LS123	X39	75188
X40	74LS273	X41	Am8253	X42	2516
X43	2516	X44	2516	X45	2516
X46	75189	X47	74LS273		

Resistors (± 5%)

R1	120	R2	480	R3	470
R4	22	R5	22	R6	240
R7	390	R8	39K	R9	1K
R10	1K	R11	1K	R12	1K
R13	1K	R14	1K	R15	1K
R16	1K	R17	1K	R18	1K
R19	22	R20	240	R21	390

Capacitors (± 20%)

C1	100 nF	C2	47 pF	C3	330 pF
C4	220 pF	C5	120 pF		

Table 10.2.a. Parts List of Central Processing Unit.

Transistors

TR1	2N2905	TR2	2N2906

Crystal

XTAL1	4MHz

Table 10.2.b. Parts List of Central Processing Unit.

Hex Address	Function
6FE0	Frame Length Register.
6FF0	Command Word Write.
6FF2	FIFO Write.
6FF4	FIFO Read.
6FF6	Control and Status Register.
6FF8	Initiate Command.
6FFA	Command Word Read.
6FFC	Interrupt Flip Flop.
6FFE	Reset Interface.

Table 10.3. 1553B Interface Memory Addresses.

Integrated Circuits

X1	74LS244	X2	74LS244	X3	Am2812
X4	Am2812	X5	74LS374	X6	74LS374
X7	74LS138	X8	74LS165	X9	74LS165
X10	74LS04	X11	74LS08	X12	74LS32
X13	74LS30	X14	74LS165	X15	74LS165
X16	74LS00	X17	74LS244	X18	74LS244
X19	74LS02	X20	74LS08	X21	74LS74
X22	74LS157	X23	74LS11	X24	74LS74
X25	74LS74	X26	74LS174	X27	9324
X28	74LS193	X29	74LS00	X30	74LS279
X31	74LS74	X32	74LS08	X33	74LS123
X34	74LS08	X35	74LS74	X36	74LS04
X37	74LS193	X38	74LS00	X39	74LS02
X40	74LS32	X41	74LS74	X42	74LS154
X43	74LS260	X44	74LS193	X45	DIL SWITCH
X46	74LS74	X47	74LS11	X48	74LS74
X49	74LS74	X50	74LS244	X51	74LS74
X52	74LS08	X53	74LS74	X54	74LS74
X55	74LS08	X56	74LS74	X57	74LS157
X58	DIL SWITCH				

Resistors (± 5%)

R1 - R16	IK
R17	150K
R18	IOK
R19 - R24	I K

Capacitors (± 20%)

C1	IOnF
C2	100pF

Table 10.4. Parts List of 1553B/Microprocessor Interface Board 1.

Integrated Circuits

X1	74LS00	X2	75452	X3	HA2522
X4	HA4905	X5	74LS00	X6	-
X7	15530	X8	74LS124	X9	74LS374
X10	74LS157	X11	74LS164	X12	9324
X13	74LS164	X14	74LS164	X15	74LS08
X16	74LS04	X17	74LS00	X18	74LS00
X19	74LS74	X20	74LS00	X21	-
X22	74LS374	X23	74LS374	X24	74LS374
X25	74LS32				

Resistors (± 5%)

R1	10K	R2	10K	R3	-
R4	47	R5	270	R6	1K
R7	10	R8	27	R9	22K
R10	22K	R11	47	R12	270
R13	4K7	R14	-	R15	10K
R16	10K	R17	1K	R18	2K2
R19	22	R20	22	R21	2K2
R22	2K2	R23	10K	R24	10K
R25	10K	R26	1K	R27	1K
R28	1K	R29	1K		

Capacitors (± 20%)

C1	10pF	C2	100pF	C3	100pF
C4	680pF	C5	68pF	C6	10pF
C7	10pF				

Diodes

D1 - D4	IN916
D5 - D8	CO46

Table 10.5.a. Parts List of 1553B/Microprocessor Interface Board 2.

Transistors

TR1	2N2905A	TR2	2N2905A	TR3	2N2221A

Crystals

XTAL1	12MHz

Transformer

T1	DDC25679

Table 10.5.b. Parts List of 1553B/Microprocessor Interface Board 2.

C4	C3	C2	C1	C0	Frame length (Bit Periods)
0	0	1	0	1	6
0	0	1	1	0	7
0	0	1	1	1	8
0	1	0	0	0	9
0	1	0	0	1	10
0	1	0	1	0	11
0	1	0	1	1	12
0	1	1	0	0	13
0	1	1	0	1	14
0	1	1	1	0	15
0	1	1	1	1	16
1	0	0	0	0	17
1	0	0	0	1	18
1	0	0	1	0	19
1	0	0	1	1	20
1	0	1	0	0	21
1	0	1	0	1	22
1	0	1	1	0	23
1	0	1	1	1	24
1	1	0	0	0	25
1	1	0	0	1	26
1	1	0	1	0	27
1	1	0	1	1	28
1	1	1	0	0	29
1	1	1	0	1	30
1	1	1	1	0	31
1	1	1	1	1	32

Table 10.6.a. Frame Length Adjustment.

Data Bit Number	Title	Function
6	DECODER PARITY	A logical '1' sets even parity
5	ENCODER PARITY	A logical '1' sets odd parity
4	C4	These bits set the frame length as overleaf.
3	C3	
2	C2	
1	C1	
0	C0	

Table 10.6.b. Frame Length Adjustment.

Integrated Circuits.

X1	74LS00	X2	75452	X3	HA2522
X4	HA4905	X5	74LS00	X6	-
X7	15531	X8	74LS124	X9	74LS30
X10	74LS138	X11	74LS374	X12	74LS374
X13	74LS157	X14	74LS157	X15	9324
X16	74LS164	X17	74LS00	X18	74LS08
X19	74LS04	X20	74LS00	X21	74LS00
X22	74LS74	X23	74LS00	X24	74LS374
X25	74LS374	X26	74LS374	X27	74LS32
X28	74LS123	X26	74LS125		

Resistors (± 5%)

R1	10K	R2	10K	R3	-
R4	47	R5	270	R6	IK
R7	10	R8	27	R9	22K
R10	22K	R11	47	R12	270
R13	4K7	R14	-	R15	10K
R16	10K	R17	1K	R18	2K2
R19	22	R20	22	R21	2K2
R22	2K2	R23	10 K	R24	10K
R25	10K	R26	IK	R27	1K
R28	1K	R29	IK	R30	1K
R31	1K	R32	9K1	R33	1K
R34	1K				

Capacitors (± 20%)

C1	10pF	C2	100pF	C3	100pF
C4	680pF	C5	68pF	C6	10pF
C7	10pF	C8	1 uF		

Table 10.7.a. Parts List of 1553B Protocol Fault Injection Board.

Diodes

D1 - D4	IN916
D5 - D8	CO46

Transistors

TR1	2N2905A	TR2	2N2905A	TR3	2N2221A

Crystals

XTAL1	12MHz.

Transformer

T1	DDC25679

Table 10.7.b. Parts List of 1553B Protocol Fault Injection Board.

1. Software Interrupt if vector not set.
2. Jump Relative to Program Counter.
3. Call subroutine relative to Program Counter.
4. Call subroutine with direct address.
5. Unimplemented instruction.
6. Invalid instruction (known action).
7. Invalid instruction (unknown action).
8. Load Program Counter and Status Word.
9. Halt
10. POP stack
11. PUSH stack.
12. Jump to direct address.

Table 11.1. Major Causes of Microprocessor System Crash.

Integrated circuits

X1	74LS32	X2	74LS00	X3	74LS32
X4	74LS08	X5	MM2114	X6	MM2114
X7	MM2114	X8	MM2114	X9	MM2114
X10	MM2114	X11	MM2114	X12	MM2114

Table 12.1. Parts List of 4K Memory Board.